Machines et langages pour traiter les ensembles de données (Textes, tableaux, fichiers)

Jean Rohmer

THÈSE

présentée à

L'UNIVERSITÉ SCIENTIFIQUE ET MÉDICALE
L'INSTITUT NATIONAL POLYTECHNIQUE
de Grenoble

pour obtenir

LE GRADE DE DOCTEUR D'ÉTAT ES SCIENCES

par

Jean ROHMER
Ingénieur ENSIMAG

Sujet de la thèse :

MACHINES ET LANGAGES POUR TRAITER LES ENSEMBLES DE DONNÉES (Textes, Tableaux, Fichiers)

Soutenue le 18 Décembre 1980 devant la Commission d'Examen composée de :

MM.	PAIR	Président
	ANCEAU BOITET DELOBEL GELENBE POUZIN	Examinateurs

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Monsieur Gabriel CAU : Président
Monsieur Joseph KLEIN : Vice-Président

MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.

PROFESSEURS TITULAIRES

MM.	AMBLARD Pierre	Clinique de dermatologie
	ARNAUD Paul	Chimie
	ARVIEU Robert	I.S.N.
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale A
	BEAUDOING André	Clinique de pédiatrie et puériculture
	BELORIZKY Elie	Physique
	BARNARD Alain	Mathématiques pures
Mme	BERTRANDIAS Françoise	Mathématiques pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques pures
	BEZES Henri	Clinique chirurgicale et traumatologie
	BLAMBERT Maurice	Mathématiques pures
	BOLLIET Louis	Informatique (I.U.T. B)
	BONNET Jean-Louis	Clinique ophtalmologie
	BONNET-EYMARD Joseph	Clinique hépato-gastro-entérologie
Mme	BONNIER Marie-Jeanne	Chimie générale
MM.	BOUCHERLE André	Chimie et toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques appliquées
	BOUTET DE MONVEL Louis	Mathématiques pures
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologique
	CALAS François	Anatomie
	CARLIER Georges	Biologie végétale
	CARRAZ Gilbert	Biologie animale et pharmacodynamie

.../...

MM.	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques pures
	CHARACHON Robert	Clinique ot-rhino-laryngologique
	CHATEAU Robert	Clinique de neurologie
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	COUDERC Pierre	Anatomie pathologique
	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumophtisiologie
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DODU Jacques	Mécanique appliquée (I.U.T. I)
	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	FONTAINE Jean-Marc	Mathématiques pures
	GAGNAIRE Didier	Chimie physique
	GALVANI Octave	Mathématiques pures
	GASTINEL Noël	Analyse numérique
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	KAHANE André	Physique générale
	KLEIN Joseph	Mathématiques pures
	KOSZUL Jean-Louis	Mathématiques pures
	KRAVTCHENKO Julien	Mécanique
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
Mme	LAJZEROWICZ Janine	Physique
MM.	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre	Mathématiques appliquées
	LEDRU Jean	Clinique médicale B
	LE ROY Philippe	Mécanique (I.U.T. I)

.../...

MM.	LLIBOUTRY Louis	Géophysique
	LOISEAUX Jean-Marie	Sciences nucléaires
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LOUP Jean	Géographie
Mlle	LUTZ Elisabeth	Mathématiques pures
MM.	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Clinique cardiologique
	MAYNARD Roger	Physique du solide
	MAZARE Yves	Clinique Médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	NEGRE Robert	Mécanique
	NOZIERES Philippe	Spectrométrie physique
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY-PEYROULA Jean-Claude	Physique
	PERRET Jean	Séméiologie médicale (neurologie)
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REVOL Michel	Urologie
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-Chirurgie
	SARRAZIN Roger	Clinique chirurgicale B
	SEIGNEURIN Raymond	Microbiologie et hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction mécanique (I.U.T. I)
	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique nucléaire
Mme	VERAIN Alice	Pharmacie galénique
MM.	VERAIN André	Physique biophysique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale

.../...

PROFESSEURS ASSOCIES

MM.	CRABBE Pierre	CERMO
	SUNIER Jules	Physique

PROFESSEURS SANS CHAIRE

Mlle	AGNIUS-DELORS Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBROISE-THOMAS Pierre	Parasitologie
	ARMAND Gilbert	Géographie
	BENZAKEN Claude	Mathématiques appliquées
	BIAREZ Jean-Pierre	Mécanique
	BILLET Jean	Géographie
	BOUCHET Yves	Anatomie
	BRUGEL Lucien	Energétique (I.U.T. I)
	BUISSON René	Physique (I.U.T. I)
	BUTEL Jean	Orthopédie
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CONTE René	Physique (I.U.T. I)
	DELOBEL Claude	M.I.A.G.
	DEPASSEL Roger	Mécanique des fluides
	GAUTRON René	Chimie
	GIDON Paul	Géologie et minéralogie
	GLENAT René	Chimie organique
	GROULADE Joseph	Biochimie médicale
	HACQUES Gérard	Calcul numérique
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et médecine préventive
	IDELMAN Simon	Physiologie animale
	JOLY Jean-René	Mathématiques pures
	JULLIEN Pierre	Mathématiques appliquées
Mme	KAHANE Josette	Physique
MM.	KRAKOWIACK Sacha	Mathématiques appliquées
	KUHN Gérard	Physique (I.U.T. I)
	LUU DUC Cuong	Chimie organique - pharmacie
	MICHOULIER Jean	Physique (I.U.T. I)
Mme	MINIER Colette	Physique (I.U.T. I)

.../...

MM.	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et minéralogie
	PFISTER Jean-Claude	Physique du solide
Mlle	PIERY Yvette	Physiologie animale
MM.	RAYNAUD Hervé	M.I.A.G.
	REBECQ Jacques	Biologie (CUS)
	REYMOND Jean-Charles	Chirurgie générale
	RICHARD Lucien	Biologie végétale
Mme	RINAUDO Marguerite	Chimie macromoléculaire
MM.	SARROT-REYNAULD Jean	Géologie
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale
MM.	STIEGLITZ Paul	Anesthésiologie
	VIALON Pierre	Géologie
	VAN CUTSEM Bernard	Mathématiques appliquées

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

MM.	ARMAND Yves	Chimie (I.U.T. I)
	BACHELOT Yvan	Endocrinologie
	BARGE Michel	Neuro-chirurgie
	BEGUIN Claude	Chimie organique
Mme	BERIEL Hélène	Pharmacodynamie
MM.	BOST Michel	Pédiatrie
	BOUCHARLAT Jacques	Psychiatrie adultes
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BRODEAU François	Mathématiques (I.U.T. B) (Personne étrangère habilitée à être directeur de thèse)
	BERNARD Pierre	Gynécologie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHARDON Michel	Géographie
	CHERADAME Hervé	Chimie papetière
	CHIAVERINA Jean	Biologie appliquée (EFP)
	COLIN DE VERDIERE Yves	Mathématiques pures
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculaire
	CORDONNER Daniel	Néphrologie
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie

.../...

MM.	CYROT Michel	Physique du solide
	DENIS Bernard	Cardiologie
	DOUCE Roland	Physiologie végétale
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	FAURE Gilbert	Urologie
	GAUTIER Robert	Chirurgie générale
	GIDON Maurice	Géologie
	GROS Yves	Physique (I.U.T. I)
	GUIGNIER Michel	Thérapeutique
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	JALBERT Pierre	Histologie
	JUNIEN-LAVILLAVROY Claude	O.R.L.
	KOLODIE Lucien	Hématologie
	LE NOC Pierre	Bactériologie-virologie
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et médecine préventive
	MALLION Jean-Michel	Médecine du travail
	MARECHAL Jean	Mécanique (I.U.T. I)
	MARTIN-BOUYER Michel	Chimie (CUS)
	MASSOT Christian	Médecine interne
	NEMOZ Alain	Thermodynamique
	NOUGARET Marcel	Automatique (I.U.T. I)
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (I.U.T. B) (Personnalité étrangère habilitée à être directeur de thèse)
	PEFFEN René	Métallurgie (I.U.T. I)
	PERRIER Guy	Géophysique-glaciologie
	PHELIP Xavier	Rhumatologie
	RACHALL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RAMBAUD Pierre	Pédiatrie
	RAPHAEL Bernard	Stomatologie
Mme	RENAUDET Jacqueline	Bactériologie (pharmacie)
MM.	ROBERT Jean-Bernard	Chimie-physique
	ROMIER Guy	Mathématiques (I.U.T. B) (Personnalité étrangère habilitée à être directeur de thèse)
	SAKAROVITCH Michel	Mathématiques appliquées

.../...

MM.	SCHAERER René	Cancérologie
Mme	SEIGLE-MURANDI Françoise	Crytogamie
MM.	STOEBNER Pierre	Anatomie pathologie
	STUTZ Pierre	Mécanique
	VROUSOS Constantin	Radiologie

MAITRES DE CONFERENCES ASSOCIES

MM.	DEVINE Roderick	Spectro Physique
	KANEKO Akira	Mathématiques pures
	JOHNSON Thomas	Mathématiques appliquées
	RAY Tuhina	Physique

MAITRE DE CONFERENCES DELEGUE

M.	ROCHAT Jacques	Hygiène et hydrologie (pharmacie)

Fait à Saint Martin d'Hères, novembre 1977

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1979-1980

Président : M. Philippe TRAYNARD

Vice-Présidents : M. Georges LESPINARD
M. René PAUTHENET

PROFESSEURS DES UNIVERSITES

MM.	ANCEAU François	Informatique fondamentale et appliquée
	BENOIT Jean	Radioéléctricité
	BESSON Jean	Chimie Minérale
	BLIMAN Samuel	Electronique
	BLOCH Daniel	Physique du Solide - Cristallographie
	BOIS Philippe	Mécanique
	BONNETAIN Lucien	Génie Chimique
	BONNIER Etienne	Métallurgie
	BOUVARD Maurice	Génie Mécanique
	BRISSONNEAU Pierre	Physique des Matériaux
	BUYLE-BODIN Maurice	Electronique
	CHARTIER Germain	Electronique
	CHERADAME Hervé	Chimie Physique Macromoléculaires
Mme	CHERUY Arlette	Automatique
MM.	CHIAVERINA Jean	Biologie, Biochimie, Agronomie
	COHEN Joseph	Electronique
	COUMES André	Electronique
	DURAND Francis	Métallurgie
	DURAND Jean-Louis	Physique Nucléaire et Corpusculaire
	FELICI Noël	Electrotechnique
	FOULARD Claude	Automatique
	GUYOT Pierre	Métallurgie Physique
	IVANES Marcel	Electrotechnique
	JOUBERT Jean-Claude	Physique du Solide - Cristallographie
	LACOUME Jean-Louis	Géographie - Traitement du Signal
	LANCIA Roland	Electronique - Automatique
	LESIEUR Marcel	Mécanique
	LESPINARD Georges	Mécanique
	LONGEQUEUE Jean-Pierre	Physique Nucléaire Corpusculaire
	MOREAU René	Mécanique
	MORET Roger	Physique Nucléaire Corpusculaire
	PARIAUD Jean-Charles	Chimie - Physique
	PAUTHENET René	Physique du Solide - Cristallographie
	PERRET René	Automatique

.../...

MM.	PERRET Robert	Electrotechnique
	PIAU Jean-Michel	Mécanique
	PIERRARD Jean-Marie	Mécanique
	POLOUJADOFF Michel	Electrotechnique
	POUPOT Christian	Electronique - Automatique
	RAMEAU Jean-Jacques	Chimie
	ROBERT André	Chimie Appliquée et des matériaux
	ROBERT François	Analyse numérique
	SABONNADIERE Jean-Claude	Electrotechnique
Mme	SAUCIER Gabrielle	Informatique fondamentale et appliquée
M.	SOHM Jean-Claude	Chimie - Physique
Mme	SCHLENKER Claire	Physique du Solide - Cristallographie
MM.	TRAYNARD Philippe	Chimie - Physique
	VEILLON Gérard	Informatique fondamentale et appliquée
	ZADWORNY François	Electronique

CHERCHEURS DU C.N.R.S. (Directeur et Maître de Recherche)

M.	FRUCHART Robert	Directeur de Recherche
MM.	ANSARA Ibrahïm	Maître de Recherche
	BRONOEL Guy	Maître de Recherche
	CARRE René	Maître de Recherche
	DAVID René	Maître de Recherche
	DRIOLE Jean	Maître de Recherche
	KAMARINOS Georges	Maître de Recherche
	KLEITZ Michel	Maître de Recherche
	LANDAU Ioan-Doré	Maître de Recherche
	MERMET Jean	Maître de Recherche
	MUNIER Jacques	Maître de Recherche

Personnalités habilitées à diriger des travaux de recherche (décision du Conseil Scientifique)

E.N.S.E.E.G.

MM.	ALLIBERT Michel	
	BERNARD Claude	
	CAILLET Marcel	
Mme	CHATILLON Catherine	
MM.	COULON Michel	
	HAMMOU Abdelkader	
	JOUD Jean-Charles	
	RAVAINE Denis	
	SAINFORT	C.E.N.G.

.../...

MM. SARRAZIN Pierre
SOUQUET Jean-Louis
TOUZAIN Philippe
URBAIN Georges — Laboratoire des Ultra-Réfractaires ODEILLO

E.N.S.M.E.E.

MM. BISCONDI Michel
BOOS Jean-Yves
GUILHOT Bernard
KOBILANSKI André
LALAUZE René
LANCELOT Françis
LE COZE Jean
LESBATS Pierre
SOUSTELLE Michel
THEVENOT François
THOMAS Gérard
TRAN MINH Canh
DRIVER Julian
RIEU Jean

E.N.S.E.R.G.

MM. BOREL Joseph
CHEHIKIAN Alain
VIKTOROVITCH Pierre

E.N.S.I.E.G.

MM. BORNARD Guy
DESCHIZEAUX Pierre
GLANGEAUD François
JAUSSAUD Pierre
Mme JOURDAIN Geneviève
MM. LEJEUNE Gérard
PERARD Jacques

E.N.S.H.G.

M. DELHAYE Jean-Marc

E.N.S.I.M.A.G.

MM. COURTIN Jaçques
LATOMBE Jean-Claude
LUCAS Michel
VERDILLON André

Je tiens à remercier :

- Monsieur le Professeur PAIR, président de l'Institut National Polytechnique de Lorraine, qui m'a fait l'honneur d'accepter de présider le jury et d'être rapporteur de cette thèse. Ses remarques ont beaucoup contribué à améliorer le fond et la forme de ce travail.

- Monsieur ANCEAU, Professeur à l'Ensimag. Cette thèse doit beaucoup à ses relectures détaillées, et plus généralement à ses efforts pour faire de l'Architecture d'Ordinateurs une discipline scientifique à part entière.

- Monsieur CELENBE, Professeur à l'Université Paris-Sud, dont les observations m'ont été fort utiles.

- Monsieur BOITET, Professeur à l'USMG, Monsieur DELOBEL, Professeur à l'USMG, Monsieur POUZIN du CNET, qui m'ont fait l'honneur de participer au jury. Je remercie en particulier M. BOITET pour son travail d'éditeur.

Je veux aussi citer mes amis du groupe CACTUS : outre François ANCEAU, Antonio ALABAU, Serge GUIBOUD-RIBAUD et Gérard NOGUEZ. Avec eux, j'ai partagé l'aventure de l'organisation des Ecoles d'Eté du Forez et de nos réflexions sur l'architecture intégrée des systèmes. Qu'ils se sentent ici remerciés collectivement et individuellement de tous les bienfaits que j'ai pu retirer de l'"esprit de Chalmazel".

Je remercie aussi :

- Monsieur GUALINO, Directeur de l'aide à la recherche de l'ADI, et Monsieur KALFON, de la société SINTRA, dont les conseils et la confiance m'ont toujours soutenu.

- Monsieur Bruno GAUDEUL, de la société R2E. A l'INRIA, j'ai pu apprécier sa gentillesse et ses dons pédagogiques en électronique.

Enfin, j'exprime ma gratitude à Monsieur IGLESIAS, Monsieur MALLET, Lydia PAGANINI et Brigitte TATARIAN pour leur sympathie et leur efficacité.

A ma famille, mes parents, ma soeur.

A mes enfants, Peggy et Adrien, qui, comme me disait jadis un ami, sont bien ce que l'on fait de mieux dans la vie...

A Nicole, pour tout, et le reste.

Quelques citations célèbres :

"Je trouve vite, mais il faut que je cherche longtemps"
(un filtre séquentiel)

"Avec des Scies, on mettrait Paris en bouteille"
(un algorithme de compactage mémoire)

"Les buffers, c'est comme les allumettes, ça ne sert qu'une fois"
(un fichier séquentiel)

P L A N

TABLE DES MATIERES

CHAPITRE 3

CHAPITRE 4

CHAPITRE 5

CHAPITRE 6

CHAPITRE 7

CHAPITRE 8

CHAPITRE 8

ANNEXES

INTRODUCTION

INTRODUCTION

Cette thèse s'intéresse au traitement de données résidant sur des supports dont l'accès est soit totalement séquentiel (cas de données diffusées par la télévision [34]), soit partiellement séquentiel (cas des disques magnétiques et optiques)

Nous nous intéresserons à la fois aux structures de machines et aux algorithmes permettant de les utiliser, les deux domaines étant évidemment étroitement liés.

Voici une série de remarques qui situent notre vision des problèmes et de la manière de les aborder.

0 1 LES SYSTEMES D'ENTREES-SORTIES OU LA FIN SACRIFIEE AUX MOYENS

La nécessité d'augmenter l'efficacité du traitement des ''fichiers'' en informatique est évidente depuis longtemps. Selon certains auteurs, 50 pour cent du temps de tous les ordinateurs du monde entier est consacré à trier ou fusionner des fichiers...

On peut résumer les efforts pour améliorer les liaisons ordinateur-fichier comme étant une tentative de RAPPROCHEMENT dans l'espace entre les fonctions de TRAITEMENT et celles de MEMORISATION.

Traditionnellement, on appelle ENTREE/SORTIE l'opération consistant à transférer une zone de données entre un périphérique (nous parlerons d'un disque) et la mémoire centrale d'un ordinateur.

La solution la plus simple a consisté à faire accomplir ce travail par un PROGRAMME de la machine , sous le nom d'entrées-sorties programmées.

Plus tard, ce travail a été confié à la MICROMACHINE, d'où le nom d'entrées-sorties microprogrammées.

Aujourd'hui, les ordinateurs puissants disposent de micromachines spécialisées dans les entrées-sorties, appelées parfois unités d'échange. Elles transfèrent directement les données entre le disque et la mémoire centrale sans passer par l'unité centrale, par une technique appelée en conséquence Accès Direct Mémoire (ADM).

L'''intelligence'' a donc quitté l'unité centrale pour se rapprocher des données.

Il reste que cette intelligence est employée à faire un simple TRANSFERT, alors que le but est un TRAITEMENT.

Ceci est un très bon exemple d'une situation fréquente en informatique: on substitue les MOYENS à la FIN. En effet:

1) la stratégie est de TRAITER les données

2) la tactique utilisée est de commencer par les transférer en mémoire, puis de les traiter.

3) on se concentre alors sur l'amélioration de ce TRANSFERT, pour aboutir comme expliqué aux unités d'échange d'aujourd'hui, très rapides et aussi sophistiquées que les ordinateurs (elles sont d'ailleurs souvent faites avec les mêmes micromachines programmées différemment)

Ces unitès d'ECHANGE sont si performantes qu'elles seraient très capables d'effectuer une bonne partie des TRAITEMENTS, qui constituaient le but initial. Mais personne ne s'en aperçoit, surtout pas les constructeurs d'unitès d'èchange.

La bonne solution aujourd'hui est pourtant de construire des unitès de TRAITEMENT DE DONNEES, qui ne transfèreraient pas dans la mèmoire des donnèes brutes, mais des donnèes partiellement ou totalement traitèes.
De telles machines devront être conçues pour s'adapter aux caractèristiques propres des donnèes à traiter (sèquentialitè, dèbit,formats).

0 2 L'EXPLOSION DES DONNEES NON NUMERIQUES

Cette explosion a lieu à la fois:

a)- en termes de quantitè:

Nous classons les donnèes ''informatisables'' en trois catègories:

* les donnèes INFORMATIQUES (nombres binaires, adresses, descripteurs) qui n'ont de sens qu'à l'intèrieur d'un ordinateur, et qui ne doivent leur existence qu'à celle des ordinateurs.

* les donnèes ADMINISTRATIVES (numèro de sècuritè sociale, donnèes de gestion) . Elles sont partiellement comprèhensibles par l'homme, mais doivent satisfaire des contraintes de taille, de format imposèes par l'ordinateur.

* les donnèes HUMAINES (textes, images, musiques.). Ce sont des donnèes qui existent INDEPENDAMMENT de l'ordinateur, depuis des siècles èventuellement. Elles reprèsentent de très loin la plus grosse masse de donnèes disponibles, mais ces donnèes HUMAINES et l'ORDINATEUR n'ont

pas été faits l'un pour l'autre ...

b)- en termes de progrès technologiques

Si l'ordinateur n'est pas adapté au TRAITEMENT des données d'origine HUMAINE, par contre les progrès de la TECHNOLOGIE de mémorisation et de transmission des données ont parfaitement suivi, voire précédé, la prolifération de l'information.

A très court terme, chaque foyer disposera de disques optiques contenant UN MILLIARD de caractères, pour le même prix que les disques microsillon actuels. Les appareils le lecture de ces disques seront eux aussi du même prix que les chaines haute fidélité.

De plus, ces disques auront un débit de l'ordre de 10 à 20 millions de caractères par seconde. Et les FIBRES OPTIQUES sont déjà prêtes à transporter de tels débits. [78] [79].

Il faut bien constater que l'élément technologique le plus à la traine dans ce domaine est tout simplement l'ordinateur ...

Actuellement, aucun ordinateur, même les plus gros ordinateurs de gestion, n'est capable d'absorber un tel débit d'informations... Et les efforts des architectes d'ordinateurs sonr essentiellement portés vers la MINIATURISATION des machines plutot que vers leur remise en question. En conséquence, les ordinateurs ne cessent pas de PRENDRE DU RETARD SUR LES DONNEES.

Cette situation est un autre exemple des mauvaises conséquences du manque d'esprit de synthése dans la conception des systémes informatiques.

Sur le plan du logiciel, il est clair que les choses doivent également changer. Il ne faudrait pas que les banques de données optiques familiales de 1990 soient accessibles uniquement via un compilateur COBOL ...

Il faut à la fois remettre en cause les langages d'interrogation et les structures des données.

L'adressage des données devra de plus en plus être

ASSOCIATIF par LE CONTENU. De même, il faudra - entre autres - tolérer les fautes d'orthographe.

0 3 UNE SOLUTION : CONSIDERER LES DONNEES COMME UN LANGAGE

Revenons à des considérations à plus court terme, et plus proches des techniques de l'informatique.

Dans le cadre du projet SCD, le problème de la conception d'une machine pour l'interrogation de l'annuaire téléphonique nous fut soumis fin 1976.

J'ai proposé à cette occasion la solution suivante:

- les données à traiter seront considérées comme des PHRASES D'UN LANGAGE

- les questions posées seront considérées comme des GRAMMAIRES

- l'interrogation de ces données revient à ne retenir que les phrases satisfaisant la grammaire.

Une manière d'expliciter cette stratégie à un informaticien est la suivante:

Soit un centre de calcul dont la bibliothèque contient un certain nombre de programmes source écrits en différents langages. On veut savoir quels sont les programmes écrits en COBOL.

Une solution très simple est de compiler TOUS les programmes source avec le compilateur COBOL. Ceux qui sont écrits en COBOL sont ceux pour lequel le compilateur a diagnostiqué ZERO ERREURS !

Si l'on est indulgent avec les programmeurs, on pourra aussi accepter comme programmes COBOL ceux ayant au plus N erreurs.

Plus gènèralement, face à un problème d'interrogation de donnèes, les questions vont être tranformèes en quelque chose de très ressemblant à un COMPILATEUR, c'est à dire à un programme capable d'ANALYSER une suite de caractères et de la TRANSFORMER en autre chose qui, dans le plus simple des cas sera une information binaire 0 ou 1, disant si la chaine considèrèe satisfait ou non à la question. Plus gènèralement, le rèsultat de cette COMPILATION pourra être une NOTE, MESURANT le degrè de satisfaction de l'enregistrement, et permettant par exemple de demander:

Donnez-moi les 10 enregistrements rèpondant le MIEUX à la question.

ou bien le rèsultat de la compilation pourra faire intervenir des notions à la sèmantique plus riche:

- projection conditionnelle sur certains champs
- cumuls, statistiques. etc...

Une fois ce principe posè, le travail consiste à analyser:

- comment adapter cette technique hèritèe du traitement des langages informatiques aux problèmes ètudiès ici, c'est à dire à la gestion des fichiers, des bases de donnèes, et à l'informatique documentaire.

- comment concevoir des machines pour effectuer le plus simplement et le plus efficacement possible cette compilation

Les résultats ont largement atteint et même dépassé nos espérances. Dans les chapitres qui suivent, nous présenterons en effet des solutions pour traiter AU VOL (en une seule passe des données et à la vitesse du disque) la plupart des problèmes d'interrogation de fichiers et de bases de données de gestion ou documentaires.

Ces solutions tournent toutes autour de la notion de FILTRE SEQUENTIEL, qui est essentiellement une machine très simple capable d'interpréter très vite des automates (d'états finis, à pile, éventuellement pseudo-parallèles), et d'effectuer des ACTIONS SEMANTIQUES au cours de leur analyse.

Une étude approfondie des aspects QUANTITATIFS de ces mécanismes de filtrage nous a montré que les besoins en vitesse et en taille mémoire des filtres sont très MODESTES, bien adaptés aux technologies d'aujourd'hui, et laissant entrevoir des performances extrêmement élevées avec les technologies de demain.

En conclusion, nous proposons des architectures de machines pour traiter les données séquentielles à la fois plus universelles, plus performantes et plus simples que toutes les autres machines de ce type déjà proposées.

Nous avons pu vérifier concrètement l'avantage d'une approche ''pluridisciplinaire'': c'est à partir de la connaissance et de l'expérience acquise en théorie des langages et en compilation que nous avons pu définir des architectures pour la gestion des données sur disque beaucoup plus efficaces que celles construites à partir d'une approche ''naive'' , trop ''matérielle'' du problème. Nous en tirons la leçon que SPECIALISE ne s'oppose pas à PROGRAMME, et que PROGRAMME ne s'oppose pas à EFFICACE

Différentes solutions sont étudiées pour permettre la tolérance à certaines FAUTES D'ORTHOGRAPHE lors du filtrage de mots.

Ensuite, un outil de programmation de filtre séquentiel est présenté. Un langage de programmation est présenté, qui permet d'une part de décrire les structures de données d'une

manière syntaxique et d'autre part de poser des questions faisant intervenir des comparaisons < = > et des conditions booléennes.

La thèse se termine par une comparaison entre APL et l'algèbre relationnelle. Ces deux langages ont tous deux la propriété intéressante d'opérer sur des ENSEMBLES, et de posséder un jeu d'opérateurs très riches. Cependant APL est orienté vers des données résidant dans des mémoires ALEATOIRES, accédées par des ADRESSES, alors que l'algèbre relationnelle adresse ses données par ASSOCIATION sur le CONTENU.

Une méthode d'interprétation de APL est proposée, et en conclusion une esquisse de fusion APL - algèbre relationnelle est présentée. Elle fait entrevoir la possibilité de concevoir un langage de programmation où l'habituelle dichotomie entre accès aux données et traitement des données serait supprimée.

CHAPITRE 1

INTRODUCTION AUX TYPES DE PROBLEMES ET D'APPLICATIONS JUSTIFIANT UNE ARCHITECTURE SPECIALISEE DANS LE TRAITEMENT DE DONNEES SEQUENTIELLES

1 INTRODUCTION AUX TYPES DE PROBLEMES ET D'APPLICATIONS JUSTIFIANT UNE ARCHITECTURE SPECIALISEE DANS LE TRAITEMENT DE DONNEES SEQUENTIELLES.

Ce chapitre examine un certain nombre d'applications de traitement de l'information où la nature logique ou physique des données est telle que les ordinateurs universels sont mal adaptés à leur traitement.

Un point commun à ces applications est que le volume des données est important. En conséquence, elles ne résident pas dans les mémoires centrales sur lesquelles les opérateurs des ordinateurs sont capables de travailler directement, mais sur des mémoires secondaires (disques), à accès séquentiel et non aléatoire, et qui doivent être raccordées à la mémoire centrale par un dispositif spécial (coupleur).

L'idée de départ est de construire une architecture adaptée à cette situation, c'est à dire:

- opérant sur le disque et non sur la mémoire

- accédant aux données de manière séquentielle et non aléatoire

La solution générale que nous proposons est de considérer les données à analyser comme le flot d'entrée d'un automate (fini ou non) et les résultats à calculer comme les sorties de cet automate, qui viendra s'interposer entre le disque et la mémoire centrale. Du fait de l'homogénéité physique entre l'entrée et la sortie de cet automate, on l'appellera aussi FILTRE.

1 1 LE TRAITEMENT DE DONNEES TEXTUELLES (D'UNE LANGUE NATURELLE)

Il s'agit d'applications comme l'informatique documentaire, l'analyse de discours, de journaux, le routage de messages.

Les données sont constituées de suites de MOTS d'une langue naturelle, assemblés de manière hiérarchique: phrases, paragraphes, chapitres ...

L'action de base à effectuer consiste à reconnaitre la présence ou l'absence d'un ou plusieurs mots-clés dans une suite de mots:

' Trouver les articles contenant les mots PARIS ou LONDRES ou BERLIN '

Ce travail est tout à fait similaire à la phase d'analyse lexicale d'un compilateur: un mécanisme d'automate fini est donc bien adapté à le réaliser.

Pour prendre en compte la structure hiérarchique des textes, il faut enrichir la notion d'automate avec celle d'appel de sous-programme -donc passer aux automates à pile- et celle de processus, pour aboutir à des automates communicants.

Le parallèle avec la compilation reste vrai: un automate reconnaissant des mots enverra ses résultats à un automate reconnaissant des phrases, tout comme l'analyse lexicale communique avec l'analyse syntaxique.

Un autre caractère spécifique aux données textuelles est la nécessité de tolérer certaines fautes d'orthographe pouvant se trouver dans les données ou les questions (ou dans les deux). Etant donné l'automate reconnaissant un mot, il sera facile de construire celui reconnaissant tous les mots qui s'en déduisent par un nombre donné de fautes d'orthographe. Cette déduction pouvant se faire soit à la compilation soit à l'interprétation de l'automate.

1 2 INTERROGATION DE FICHIERS INFORMATIQUES

Les fichiers informatiques au sens habituel du terme prèsentent des diffèrences avec les donnèes textuelles:

- les enregistrements sont fortement structurès, par exemple comme un ensemble (plat ou hièrarchique) de CHAMPS dèsignès par un nom, et ne pouvant contenir qu'une seule valeur (et non un suite de mots), appartenant à un type prèdèfini, muni souvent d'une relation d'ordre.

Il ne s'agit donc plus de rechercher l'APPARTENANCE d'un mot à un champ, mais d'effectuer des COMPARAISONS (<, =, >) entre des constantes et le contenu d'un champ.

Il faut de plus un mècanisme pour analyser la structure de l'enregistrement, c'est à dire pour reconnaitre l'emplacement des differents champs.

Enfin, la plupart des critères d'interrogation font intervenir des conditions booléennes sur les comparaisons èlèmentaires:

''Trouver le nom des personnes dont l'age est compris entre 50 et 60 ET habitant à PARIS OU MARSEILLE''

Nous dècouvrirons progressivement comment ces differents points (comparaisons < = > , reconnaissance des champs, calcul booléen) peuvent être rèalisès soit directement, soit indirectement par le seul mècanisme d'automate fini.

1 3 OPERATIONS MULTIFICHIERS SUR BASES DE DONNEES

L'analyse par automate (ou filtrage) d'un fichier peut se symboliser par l'application d'une fonction f(F, A) où F est le fichier et A la description de l'automate, et dont le résultat est lui-même un fichier.

Une base de données est constituée d'un ensemble de fichiers, et permet des opérations MULTIFICHIERS, alors que le filtrage est typiquement une opération MONOFICHIER.

La plus courante de ces opérations multifichiers est la JONCTION entre deux fichiers F1 et F2, dont la forme générale est:

$$j(F1,F2,\alpha)$$

où α précise la condition de jonction.

Nous verrons qu'il est possible de réaliser UNE jonction avec DEUX opérations de filtrage de la forme:

R1 ← f(F2, A1) filtrage du second fichier

A2 ← c(R1) compilation du fichier R1 en automate

R2 ← f(F1, A2) filtrage du premier fichier

1 4 OPERATIONS DE CALCUL GLOBAL SUR DES DONNEES SEQUENTIELLES.

Dans les exemples précédents, les traitements avaient une portée LOCALE à chaque enregistrement, et consistaient par exemple à le filtrer ou non.

Une autre classe d'applications consiste à effectuer des calculs GLOBAUX à l'ensemble des enregistrements, que ce soient des traitements simples comme des cumuls, comptages, histogrammes, ou des traitements plus complexes d'ANALYSE DE DONNEES: classification automatique, analyse de la variance, ...

Là encore, un filtrage par automate s'avère très utile; il suffit de lui adjoindre une unité de calcul vers laquelle il enverra des instructions en fonction des situations rencontrées au cours de l'analyse. Par exemple, s'il s'agit de compter le nombre d'occurences de certains mots, il suffira à l'automate de commander une incrémentation chaque fois qu'il aura atteint l'état correspondant à la reconnaissance d'un mot. La fréquence de telles ACTIONS SEMANTIQUES étant relativement faible par rapport à la fréquence d'entrée du filtre, c'est encore ici le filtre qui effectue l'essentiel du travail.

Dans les problèmes de classification, le but est de regrouper les enregistrements qui se ''ressemblent'' au sein d'un même bloc de mémoire. Une manière d'accomplir ce regroupement consiste à maintenir pour chaque bloc un ''RESUME'' des valeurs qu'il contient, de sorte que la recherche du bloc dans lequel insérer un nouvel enregistrement se fasse par un simple FILTRAGE des résumés.

1 5 REPRESENTATION ET COMPROMIS DE REALISATION DES AUTOMATES.

Nous venons de présenter un grand nombre d'exemples de problèmes pour lesquels le filtrage par automates offre une solution intuitivement satisfaisante. Nous voyons ainsi apparaitre une certaine UNIVERSALITE des applications du filtrage. Il s'agit maintenant de définir une REPRESENTATION des automates qui conduise à une architecture satisfaisante non seulement sur le plan qualitatif (ce qui est acquis d'après ce qui précède), mais aussi sur le plan QUANTITATIF.

Etant donné un alphabet A de N éléments, un automate fini

déterministe est défini par:

- un ensemble d'états E

- pour chaque état e ∈ E, un ensemble de N couples (ai, ei) indiquant, pour tout ai ∈ A, que ei est l'état successeur de e en cas de lecture du symbole ai en entrée.

En pratique, un automate sera codé dans une mémoire. A chaque état de l'automate, il faudra associer une structure de données -définie par son adresse de début- qui contiendra entre autres les adresses de début des structures de données associées aux états successeurs de l'état en cours.

Le seul but de cette structure de données est de permettre de réaliser la fonction de transition de l'état courant t(a), qui, pour tout a ∈ A, donne l'adresse d'un état suivant.

De nombreuses méthodes existent pour représenter et interpréter cette fonction.

La plus triviale consiste à constituer la liste des couples (ai, t(ai)) et à les balayer séquentiellement jusqu'à trouver la valeur recherchée.

Pour accélérer cette recherche, on peut la transformer en une recherche dichotomique, mais ceci obligera à CLASSER la liste selon les ai, donc augmentera le temps de création de cette liste.

Une troisième solution consiste à construire une table T de N adresses d'état suivant (une par élément de l'alphabet), et à utiliser le caractère ai comme un index dans cette table, avec T[ai] = t(ai).
Cette solution est d'une exécution très rapide, mais est couteuse en place mémoire.

Plus généralement, une part importante du travail qui suit est consacrée à étudier différentes représentations des

automates, en cherchant à minimiser les quantités suivantes:

- le temps nécessaire à la traduction d'une requête en automate

- la place prise en mémoire par l'automate

- le temps de parcours de l'automate, qui, en pratique, dépendra essentiellement du nombre d'accès mémoire nécessaires pour évaluer la fonction de transition

- la largeur des mots de la mémoire contenant l'automate.

Ces différentes quantités varient de manière antagoniste, et il faudra trouver des compromis acceptables. Les termes de ces compromis ne sont pas énonçables à priori, mais dépendent au contraire d'une connaissance quantitative des applications: distribution des nombres de mots-clés, de leurs longueurs, de leur fréquence dans les données et les questions, de la fréquence d'occurence des lettres dans une langue naturelle, de la complexité des expressions booléennes.

Les différentes solutions que nous proposerons seront en conséquence évaluées soit à l'aide de simulations que nous avons faites, soit à partir de mesures statistiques publiées par d'autres auteurs.

Nous arriverons à la conclusion très intéressante que, en pratique, la plupart des problèmes énoncés au début de ce chapitre pourront être résolus par des opérations de filtrage réalisées AU VOL sur les données, et ceci indépendemment de la complexité des requêtes.

CHAPITRE 2

TECHNIQUES ET COMPROMIS DE REALISATION DES FILTRES

2 TECHNIQUES ET COMPROMIS DE REALISATION DES FILTRES

2 1 LES CONTRAINTES QUALITATIVES ET QUANTITATIVES POUR LA REALISATION D'UN FILTRE

Il faut d'abord noter que l'opposition entre qualitatif et quantitatif n'est pas toujours très claire: ainsi, le choix d'un codage dichotomique de l'arbre des comparaisons entre les successeurs d'un état de l'automate sera à la fois dû au respect d'une contrainte quantitative (pouvoir suivre le débit d'entrée indépendamment du nombre d'alternatives) et d'une contrainte qualitative (pouvour faire des comparaisons < et >).

De plus, on peut classer les contraintes (qualitatives ou quantitatives) selon le domaine auxquel elles s'appliquent:

- contraintes économiques de fabrication : nombre de portes logiques, nombre de boitiers, nombre de connections avec l'extérieur

- contraintes de programmation: difficultés de la compilation des automates et temps nécessaire à cette compilation. Ceci est très important pour la réalisation des opérations de jonction des bases de données relationnelles. De plus, dans un environnement conversationnel, cela ne sert à rien d'avoir une exécution rapide d'une requête, si on perd trop de temps à la compiler.

- contraintes de vitesse d'exécution, de suivi d'un débit d'entrée.

* La plupart des disques à bras mobiles ont un débit de 10 Mbits par seconde, et le nouveau modèle IBM 3370 a un débit de 20 Mbits (il utilise la technique des films minces).

* Les disques optiques sont capables d'atteindre 100Mbits/sec, bien que les premiers modèles disponibles aient un débit très lent du fait du marché qu'ils visent: le remplacement des bandes magnétiques pour l'archivage,

* Les fibres optiques peuvent atteindre 200Mbits/sec

Le respect de ces contraintes peut être obtenu de deux manières:

* rapidité du matériel
* limitation de la complexité de la requête

Le recours au deuxième point est difficile à faire admettre à l'utilisateur (sa requête, acceptée par un système logiciel lent, est refusée par une machine spécialisée rapide ....). C'est pourquoi nous avons porté beaucoup d'efforts pour que le temps de traitement soit indépendant de la complexité de la question. Si cet objectif a été atteint pour les opérations classiques d'interrogation, il est plus difficile à conserver quand il s'agit de faire aussi du traitement. Dans ce cas, il est impossible de savoir à la compilation si la requête 'suivra le débit'. Il est donc nécessaire de prévoir dans l'architecture la possibilité de ne pas suivre temporairement le débit du disque plutot que de refuser la question à l'exécution pour cause de débordement de files d'attente.

2 2 PROPOSITION DE PLUSIEURS SOLUTIONS POUR L'IMPLEMENTATION DES FILTRES.

PLAN DU CHAPITRE

Différentes solutions ne faisant pas intervenir les automates sont d'abord présentées. Elles proviennent d'exemples réels de machines bases de données qui ont été soit effectivement réalisées, soit simplement proposées dans la littérature. Leur défaut commun par rapport aux filtres est d'être à la fois plus complexes et plus couteuses à réaliser tout en ayant moins de possibilités et de moins bonnes performances. La cause de ces défauts provient à notre avis d'une trop grande précipitation de leurs auteurs à vouloir 'câbler' à tout prix telle ou telle fonction élémentaire, sans se demander si une solution A LA FOIS PROGRAMMEE ET SPECIALISEE n' apporterait pas plus de souplesse et de généralité pour un prix inférieur. (Une leçon très générale qui peut être tirée de notre travail est que SPECIALISE ne s'oppose pas à PROGRAMMABLE et que PROGRAMMABLE ne s'oppose pas à RAPIDE .

Ensuite, les différentes solutions que nous proposons seront présentées en trois parties, chacune s'intéressant à un aspect différent du travail que peut accomplir un FS . Il est apparu que la classification la plus significative était très proche -et comment s'en étonner - de celle rencontrée dans la traduction des langages de programmation.

- aspects lexicaux : comment reconnaitre des mots ou des nombres

- aspects syntaxiques: comment reconnaitre la structure interne des données à traiter (découpage en champs par exemple)

- aspects sémantiques: calculer une fonction portant sur les données analysées, une expression booléenne par exemple.

2 3 QUELQUES SOLUTIONS DE FILTRAGE ET DE RECHERCHE PAR CONTENU N'UTILISANT PAS LES AUTOMATES

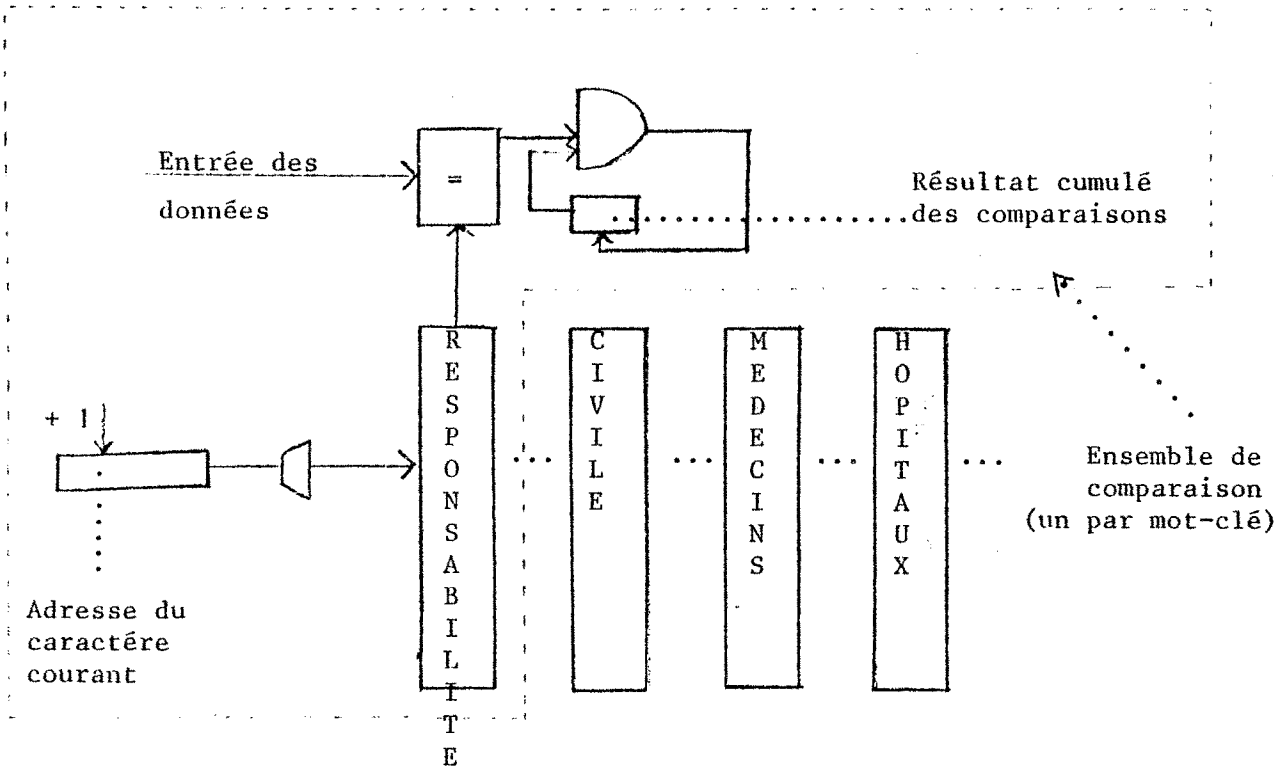
Dans tous les cas il s'agit de reconnaitre, dans un flot séquentiel de caractères, si certaines zones sont égales, inférieures ou supérieures à des constantes données, puis d'évaluer éventuellement une expression booléenne portant sur ces comparaisons élémentaires.

Par exemple:
- trouver les enregistrements contenant PARIS ou LONDRES et pas TOKYO
- trouver les personnes dont le salaire est > 500 et l'age compris entre 40 et 45

2 3 1 Comparateurs en parallèle. Exemple: la machine TGI [20]

Une solution assez brutale consiste à ranger chacune des constantes intervenant dans la question dans une mémoire distincte et à mettre en face de chacune d'elles un comparateur. et un registre d'adresse incrémenté à chaque nouveau caractère en entrée. Il faut supposer de plus que, pour chaque champ, il

existe un caractère délimiteur annonçant son arrivée.



Les auteurs donnent l'exemple suivant:

On recherche les mots'
RESPONSABILITE,CIVILE,MEDECINS,HOPITAUX,PUBLICS

Il faut donc mettre chacun d'eux dans une mémoire, et chaque fois qu'un nouveau mot se présente en entrée, sa première lettre est comparée EN PARALLELE par 5 comparateurs aux ièmes lettres de chacun des 5 mots recherchés. Pour chaque comparateur, on fait en plus un ET cumulatif du résultat des comparaisons successives. A la fin d'un mot, on sait s'il appartenait à l'ensemble recherché si et seulement si l'un des résultats cumulés des comparaisons est égal à 1.

Le gaspillage évident de cette solution est que les comparateurs travaillent la plupart du temps inutilement, puisque leur résultat cumulé est déjà à zéro et qu'il y restera ...

La raison invoquée per les auteurs est qu'ainsi on est sûr de

travailler à vitesse constante, contrairement au cas où l'on ne dispose que d'un seul comparateur, et où il faut faire des tests de manière hiérarchique, donc en nombre variable pour chaque caractère en entrée. Cet argument ne tient plus si l'on utilise un hiérarchie dichotomique; il est alors facile de borner le nombre maximum de comparaisons (3 3 2 2).

De plus, cette solution est beaucoup trop rigide:

- le nombre de mots-clès est borné par celui des ensembles de comparaison
- la taille de chaque mot est bornée par celle des mémoires individuelles les contenant
- la recherche d'une sous-chaine dans une chaine est impossible
- la tolérance aux fautes d'orthographe est impossible

La solution TGI utilise beaucoup de matériel pour des performances médiocres.

2 3 2 Opérateurs d'évaluation de monomes booléens (Exemple: la machine DBC [21]

La machine (papier) DBC est souvent considérée comme la première -bien que récente, 1977- architecture où TOUS les problémes d'implémentation d'un SGBD ont été abordés (mise à jour, clustering, protection ...et pas seulement interrogation) .

Elle est bâtie comme un ensemble de processeurs spécialisés chacun dans un de ces problémes. Pour ce qui nous intéresse ici, la recherche par contenu, sa solution n'est pas très intéressante:

on dispose d'opérateurs élémentaires, les TIP dont chacun peut seulement calculer un ET sur des comparaisons élémentaires, de la forme:

Age = 50 ET Salaire > 500 ET Nom ≠ JEAN

avec la restriction qu'un même champ ne peut apparaitre plusieurs fois comme dans Nom ≠ JULES ET Nom ≠ JEAN

Si l'on a une question comportant des OU, il faut donc la mettre sous forme normale disjonctive (un OU de ET) et disposer d'autant de TIP en parallèle qu'il y a de monomes dans la

condition. Comme pour la solution TGI, il y a ici un gaspillage évident de matériel, et un manque total de souplesse. Par exemple, l'expression

(Nom est JEAN ou JULES ou PIERRE ou LOUIS) et (Age est 50 ou 61 ou 65 ou 70) nécessite 16 TIP en parallèle... qui, comme dans TGI font la plupart du temps un travail inutile.

Remarquons au passage qu'il est ainsi facile de faire un multiprocesseur dont tous les processeurs sont actifs à tout instant, en oubliant que ce qu'ils font ne sert à rien.

2 3 3 Utilisation de grandes mémoires associatives comme STARAN [22]

Soient des enregistrements de la forme:
Nom, age, salaire, chaque champ ayant une longueur FIXE.

On dispose d'une mémoire associative de N enregistrements, et l'on peut, en un seul cycle, comparer ces N positions mémoire à un enregistrement clé donné, chargé dans un registre. Si l'on a P caractères par enregistrement, on fait donc N×P comparaisons de caractères en parallèle (sur STARAN N =1024 et P =256). Malheureusement, ceci ne sert à rien puisque il faut faire défiler par blocs successifs la mémoire secondaire dans la mémoire associative; donc le débit est celui de cette mémoire secondaire, et non celui de la mémoire associative.

Des solutions similaires ont été proposées sur la machine PROPAL [23]

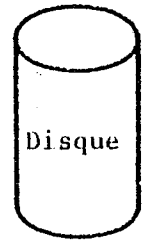
Ne parlons pas des problèmes posés par le fait que le format doit être fixe.

Le but -avoué - de ces propositions est uniquement de trouver des applications aux processeurs matriciels

Enregistrement clé :

JULES	53	2000

0	MAX	20	1000
0	HENRY	30	2000
0	LOUIS	53	2000
0	MAX	53	2000
0	LUCIEN	40	5000
0	BERTHE	20	4000
1	JULES	53	2000
0	ALEX	20	1000



Résultats de comparaison

2 3 4 Opérateurs élémentaires travaillant sur des mémoires circulantes. Exemple CASSM [24], RAP [25]

Ces architectures 'profitent' du fait que les données sont organisées en un ensemble de mémoires circulantes (CCD, RAM simulant une CCD, Disque avec lecture et écriture quasi simultanées) pour faire en plusieurs passes ce qu'un filtre fait en une seule. Par contre, elles travaillent en série (une seule comparaison à la fois, contrairement aux solutions précédentes).

Soit à évaluer (Nom = JEAN) ou (Nom = JULES). Dans une première passe, on compare le champ Nom à JEAN et on range le résultat dans une pile située à la fin de l'enregistrement (d'où la nécessité de pouvoir lire ET écrire lors de la même transaction). Au tour suivant, on compare à JULES, et on fait le OU du résultat avec le sommet de pile. A la fin, chaque enregistrement dispose du résultat de la condition booléenne au fond de sa pile. Il faut donc autant de rotations qu'il y a d'opérateurs booléens dans la comparaison.

2 4 Conclusion sur ces méthodes

Ces méthodes nécessitent toutes:
- ou bien plusieurs opérateurs de comparaison parallèles pour travailler en une seule passe
- ou bien plusieurs passes si elles n'utilisent qu'un seul opérateur de comparaison

La méthode de filtrage par automates concilie les deux: un seul comparateur et une seule passe. Quel sera le prix à payer? Il concerne la taille mémoire prise par l'automate, le débit d'entrée maximal accepté, la complexité de compilation de l'automate.

Les paragraphes suivants examinent en détail tous ces points, et proposent différents compromis de réalisation tels que ce prix à payer apparait comme négligeable, voire inexistant. Même avec les techniques les plus triviales, les inconvénients des automates sont de peu de poids face à leur avantage fondamental : la souplesse d'une solution PROGRAMMEE face à des solution inutilement CABLEES.

D'autre part, nous montrerons que la distinction habituelle entre:
- machines de recherche dans des textes peu formattés utilisant des automates d'une part
- machines de gestion de bases de données formattées utilisant des opérateurs câblés spéciaux d'autre part

est peu intéressante dans la mesure où les automates, s'ils autorisent les comparaisons $<$ et $>$ en plus du $=$ et $\neq$, conviennent parfaitement pour les bases de données formattées.

Donc, les propositions qui suivent ont, en plus de leurs avantages propres, un caractère assez universel

CHAPITRE 3

TECHNIQUES POUR RESOUDRE LES PROBLEMES LEXICAUX:

LE PARCOURS D'UN AUTOMATE FINI

3 TECHNIQUES POUR RESOUDRE LES PROBLEMES LEXICAUX : LE PARCOURS D'UN AUTOMATE FINI:

3 1 POSITION DU PROBLEME

En général, un état dans l'automate peut avoir jusqu'à N successeurs, si N est le nombre de caractères de l'alphabet utilisé. Par exemple, si l'on veut connaitre toutes les personnes habitant dans une ville appartenant à un ensemble de 500 villes données, il faut construire l'automate de reconnaissance des 500 noms de villes, et commencer par comparer la première lettre reçue avec jusqu'à 26 lettres différentes. La principale difficulté qui vient à l'esprit est: comment faire ces comparaisons le plus vite possible pour trouver l'état suivant dans l'automate avant l'arrivée du caractère suivant en entrée ?

Une fois cette contrainte respectée, il faudra essayer de minimiser le cout du matériel nécessaire à ce parcours de l'automate, donc de minimiser deux facteurs en général antagonistes:

- la taille mémoire pour représenter l'automate
- la complexité de la logique de séquencement

Enfin, il faut minimiser le temps nécessaire à la création d'un automate, soit à partir d'une question (pour les opérations unaires), soit à partir d'un fichier (pour les opérations binaires). Ce point a l'air à première vue secondaire (peu 'noble'), mais les premières expériences en vraie grandeur montrent qu'il est en fait très critique:

dans AFP [26], la compilation des automates à partir des questions est très lente, bien que ces questions soient très simplistes: on compile seulement 10 états de l'automate par seconde sur un PDP11/40.

Dans un système transactionnel ou relationnel, cette compilation peut donc être un goulot d'étranglement inattendu.

Ce dernier point nous rend un peu sceptique face à des affirmations du genre: 'les processeurs bases de données vont simplifier le logiciel [27]'. Il faudra peut-être se contenter d'améliorer les performances et de ne simplifier que certaines parties du logiciel tout en en compliquant éventuellement d'autres.

Quelques définitions.

- On considère des automates d'états finis; à chaque état on peut associer une adresse le désignant

- On appellera CE (caractère en entrée) le dernier caractère lu à l'entrée de l'automate.

- Pour chaque état de l'automate, CS (caractères successeurs) représente l'ensemble des valeurs des CE admissibles pour provoquer la transition vers un état suivant

- AS (adresses successeurs) est l'ensemble des adresses des états successeurs associés à CS

- AE (adresse d'échec) est l'adresse de l'état suivant pour le cas où CE n'appartient pas à CS

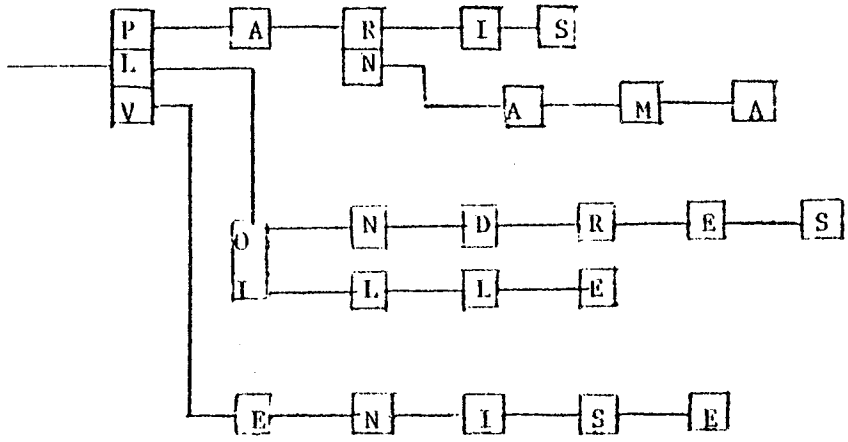
- N désigne le nombre d'élèments dans l'alphabet d'entrée

3 2 LES SOLUTIONS 'NAIVES'

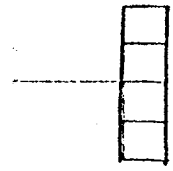
3 2 1 Balayage séquentiel des alternatives dans l'automate

Les n caractères successeurs d'un état sont rangés séquentiellement et à chacun d'eux est associée l'adresse de l'état suivant. Il faut donc en moyenne n ÷ 2 comparaisons pour trouver le successeur dans le cas où le caractère en entrée appartient à la liste et n comparaisons dans le cas contraire. De plus, il faut n+1 adresses d'état successeur, une pour chaque caractère de la liste, plus une pour tous les autres cas, si le caractère d'entrée n'appartient pas à la liste.

Exemple: Trouver les enregistrements contenant les mots PARIS ou PANAMA ou LONDRES ou LILLE ou ou VENISE.



1 état ↔ 1 trait
suivi de sa liste CS :



L'inconvénient de cette solution est que le nombre de comparaisons peut être très grand, et égal au nombre de caractères de l'alphabet utilisé.

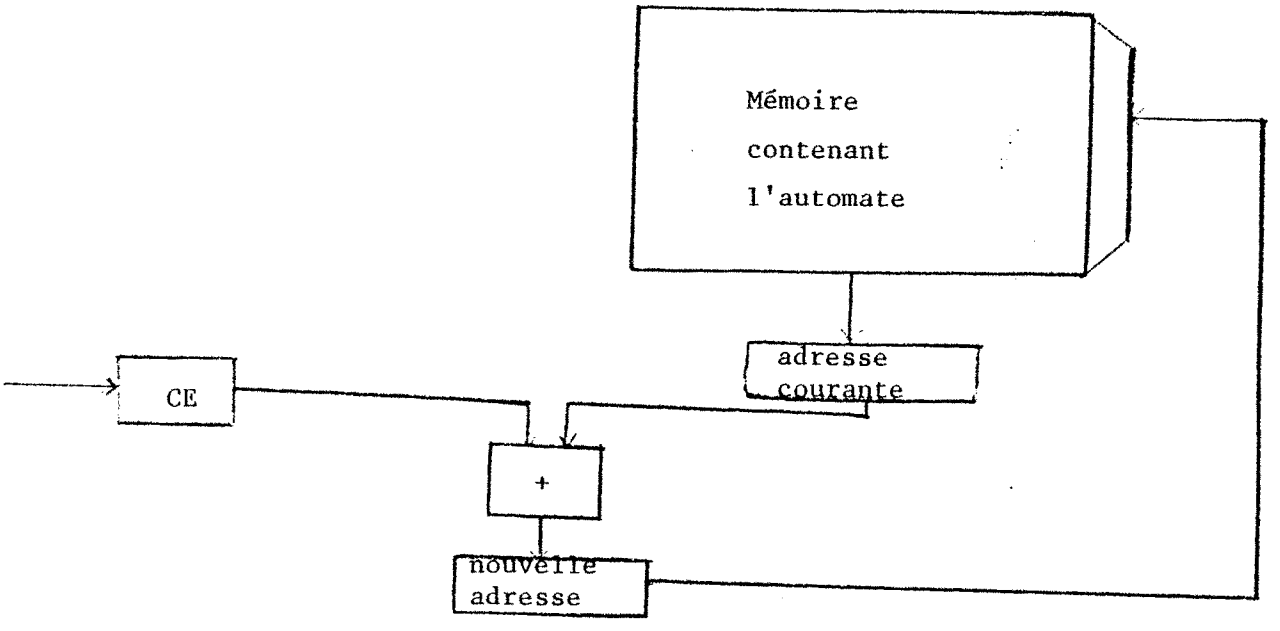
Par contre, la taille de l'automate est raisonnable et sa compilation assez simple.

A notre connaissance, cette méthode n'a jamais été utilisée dans une machine associative, par crainte de sa lenteur sans doute. Elle est cependant envisageable si on est sur que le nombre de successeurs est faible, une condition suffisante pour cela étant qu'il y ait peu de mots clés dans la question.

Ainsi, dans l'exemple précédent, il faut en moyenne 3 comparaisons pour la première lettre, 2 pour la seconde et 1 pour les suivantes. Comme un mot en Français fait en moyenne 6 lettres [28], on arrive à $(3 + 2 + 4\times1)\div6 = 1,5$ comparaisons entre deux CE, ce qui est très confortable pour suivre un disque par exemple. Cette solution n'est donc pas à rejeter à priori pour résoudre les cas les plus simples.

3 2 2 Tables d'adresse indexées par le caractère en entrée

Cette solution prend le contrepied de la précédente: à chaque état, on associe une table de N adresses de successeurs et l'on indice cette table avec le caractère en entrée convenablement codé entre 0 et N-1 pour trouver l'état suivant en UN SEUL ACCES MEMOIRE. Donc la rapidité est maximum, par contre l'occupation mémoire est énorme puisque tout état nécessite la place de N adresses en mémoire même s'il n'a que 1 seul successeur. La mise en oeuvre de cette technique est évidemment très simple:



Il est clair qu'elle est surtout intéressante si
- l'alphabet est très petit
- le nombre de successeurs d'un état est souvent proche du maximum

Ces deux facteurs s'harmonisent bien. On peut par exemple considérer un alphabet à seulement deux éléments. Alors l'automate devient binaire, et par exemple un caractère de 8 bits nécessite au maximum 8 adresses pour le représenter. Par contre, il faut faire un accès mémoire non pas tous les caractères mais TOUT LES BITS.

Exemple:

Reconnaitre : mot 1 : 10110
mot 2 : 1010
mot 3 : 010
mot 4 : 001

MEM :

0	aller à 6
1	aller à 4
2	aller à 3
3	mot 4 trouvé
4	
5	mot 3 trouvé
6	échec
7	aller à 8
8	aller à 10
9	mot 2 trouvé
10	échec
11	échec
	mot 1 trouvé

Algorithme :

adresse ← 0

Lire CE suivant
Si CE = 0 alors
Adresse ← Adresse + 1
Si CE = 1 alors
Adresse ← MEM [Adresse]

Ces solutions 'naives' ont des avantages et des inconvènients complèmentaires et peuvent être amèliorèes de plusieurs manières:

- la première peut être accélérée en faisant plusieurs comparaisons en parallèle. On tendra alors vers une solution de type TGI [20] tout en gardant la souplesse des automates.

- la seconde peut être utile si on la limite à la représentation des états ayant plus d'un successeur, ce qui est le cas le plus rare, comme noté dans [29].

Ceci conduira à notre proposition d'une solution exposée en 3 4 et qui a l'avantage supplèmentaire d'être ultra rapide à compiler, au point de permettre la réalisation d'opérations de jonction sur égalité en un TEMPS LINEAIRE.

3 3 TECHNIQUES OPTIMISEES DE REPRESENTATION DES AUTOMATES

3 3 1 LA REPRESENTATION DE BIRD [26] DANS LA MACHINE AFP

Elle repose sur un mèlange astucieux de plusieurs compromis:

a)- les caractères considérés sont de 3 ou 4 bits (petit alphabet)

b)- on ne veut mémoriser que les adresses utiles (une par AS plus une pour AE)

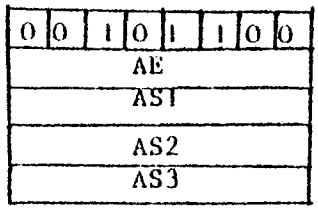
c)- on veut faire un nombre constant d'accés à la mémoire pour chaque CE

Supposons que le caractère fasse 3 bits. Il y a au plus 8 successeurs. On construit la table de présence de ces successeurs; c'est une table de 8 bits, le iéme valant 1 si et seulement si il existe une adresse d'état suivant pour cette valeur. Les adresses de successeurs sont d'autre part rangées consécutivement et dans l'ordre croissant des caractéres auxquelles elles sont associées. Supposons par exemple que les caractéres successeurs sont 2 , 4 et 5.

Alors la table de présence est 0 0 1 0 1 1 0 0

Et la liste d'adresses AS est:
- adresse pour 2
- adresse pour 4
- adresse pour 5

De plus l'adresse d'échec est précisée.



Format d'un état dans AFP

L'algorithme est alors le suivant:

- on indice la table de présence avec la valeur de CE. Si le résultat vaut zéro, alors l'adresse suivante est l'adresse d'échec. Sinon, l'adresse suivante est la Iéme dans la liste, où I est le nombre de '1' à gauche de celui que l'on vient de lire dans la table de présence.

La raison de l'utilisation de caractères courts est évidente: avec des caractères de 8 bits, la table de présence ferait 256 bits ...

Un inconvénient de cette méthode est que l'algorithme est assez complexe. (A moins de l'implémenter dans une mémoire morte de (8 + 3) mots dans le cas de caractères de 3 bits, ce qui fait de toutes façons un accès mémoire supplémentaire.)

Cet algorithme nécessite de plus 2 accès mémoire par caractère (un pour la table de présence, un pour la liste d'adresses), donc pour un caractère 'normal' de 7 ou 8 bits, il faudra 4 accès mémoire, plus une partie combinatoire assez complexe répétée deux fois. Or nous avons vu - et nous verrons encore- que l'on peut faire beaucoup de choses en quatre accès avec des algorithmes plus simples.

Un second inconvénient est que la compilation d'une structure de données aussi complexe est longue (10 noeuds par seconde sur un PDP11/40)

Enfin, par comparaison aux méthodes que nous préconisons, ces automates ne permettent pas de prendre en compte des relations d'ordre (< ≤ ≥ >). Ils sont typiquement orientés vers des applications textuelles et documentaires. Mais même dans ce domaine, il est nécessaire de travailler sur des < , > avec des critères de la forme 'Date comprise entre 1950 et 1957'. La solution de AFP est alors de dire ' Date = 1950 ou 1951 ... ou 1957' !

Le gros avantage est une bonne optimisation de la taille mémoire, d'autant plus utile ici que les auteurs veulent compiler ensemble plusieurs questions, leur capacité mémoire totale pour l'automate étant de 64K mots de 16 bits.

3 3 2 LA REPRESENTATION DICHOTOMIQUE DES AUTOMATES

3 3 2 1 Un retour aux bonnes vieilles solutions du logiciel

Le problème de savoir si un mot appartient ou non à une liste de mots se pose depuis longtemps aux écrivains de compilateurs, sous le nom de 'gestion de tables de symboles'. Pour le traiter avec efficacité, deux méthodes sont couramment utilisées:

-si la liste est connue a priori (cas des mots réservés du langage), on ordonne cette liste et on effectue une recherche dichotomique

-si le liste est inconnue à priori, (cas des identificateurs déclarés par le programmeur), on utilise les méthodes de 'hash-coding'

Ces deux techniques sont très connues et très utilisées par les spécialistes du logiciel et leurs performances ont été étudiées en détail par de nombreux auteurs [32, 33].

L'inconvénient de la première est qu'il faut faire un tri, et l'inconvénient de la deuxième est que le codage ne conserve pas la relation d'ordre. De plus, le 'hash-coding' s'effectue en deux temps:

1) calcul du hash-code (ceci peut être court: somme modulo 256 des caractères d'un mot par exemple)

2) comparaison du mot avec tous ceux ayant le même code. Ceci peut par contre être assez long, et, même si

la liste des mots ayant le même code est courte, il faut de toutes façons MEMORISER le mot arrivant (par exemple en même temps que l'on fait le calcul du code) pour pouvoir ensuite le comparer à ceux de la liste.

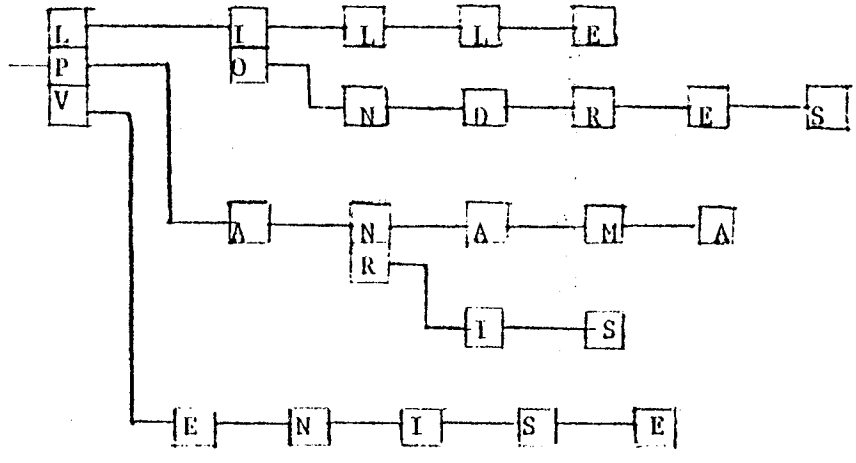
Finalement, la méthode du hash-code est assez lourde à mettre en oeuvre, assez lente, inadaptée aux comparaisons < et > , donc à exclure pour les bases de données classiques. De plus, pour les données textuelles, elle ne permet pas la recherche d'une sous-chaine dans une chaine, ni la tolérance aux fautes d'orthographe. Néanmoins, nous en proposerons une forme dégénérée utilisable dans des applications grand-public comme la réception sélective de textes télédiffusés [34].

3 3 2 2 Automates dichotomiques compilés

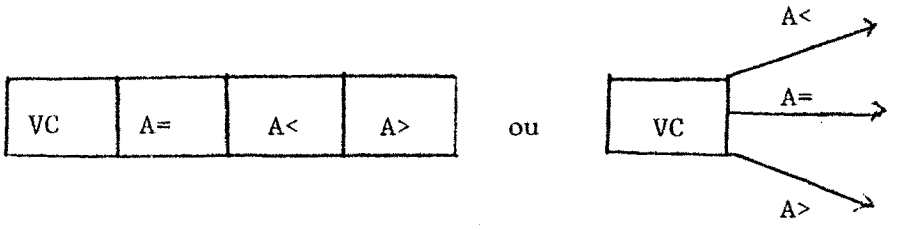
Revenons à la recherche dichotomique dans une table de symboles. Elle a l'avantage de bien s'adapter à la technique des automates d'états finis.

Ordonnons alphabétiquement la liste de mots précédente:

LILLE
LONDRES
PANAMA
PARIS
VENISE



Nous allons maintenant COMPILER l'automate dans un langage objet ne contenant qu'un seul type d'instructions de la forme:



-valeur d'un caractère VC
-adresse A=
-adresse A<
-adresse A>

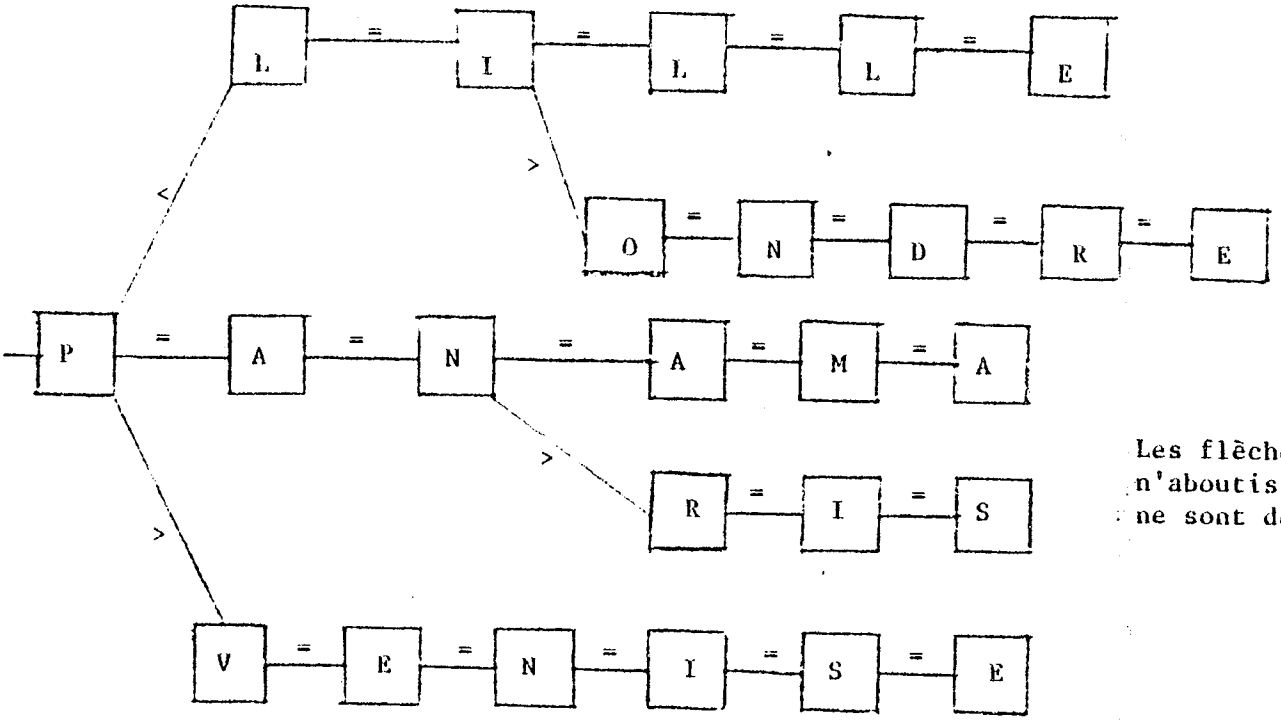
et dont la sémantique est la suivante:

-si le caractère en entrée est égal à VC, alors
  * lire CE suivant en entrée
  * aller à l'adresse A=

-si le caractère en entrée est inférieur à VC
  * aller à l'adresse A<
  * (ne pas lire le CE suivant en entrée)

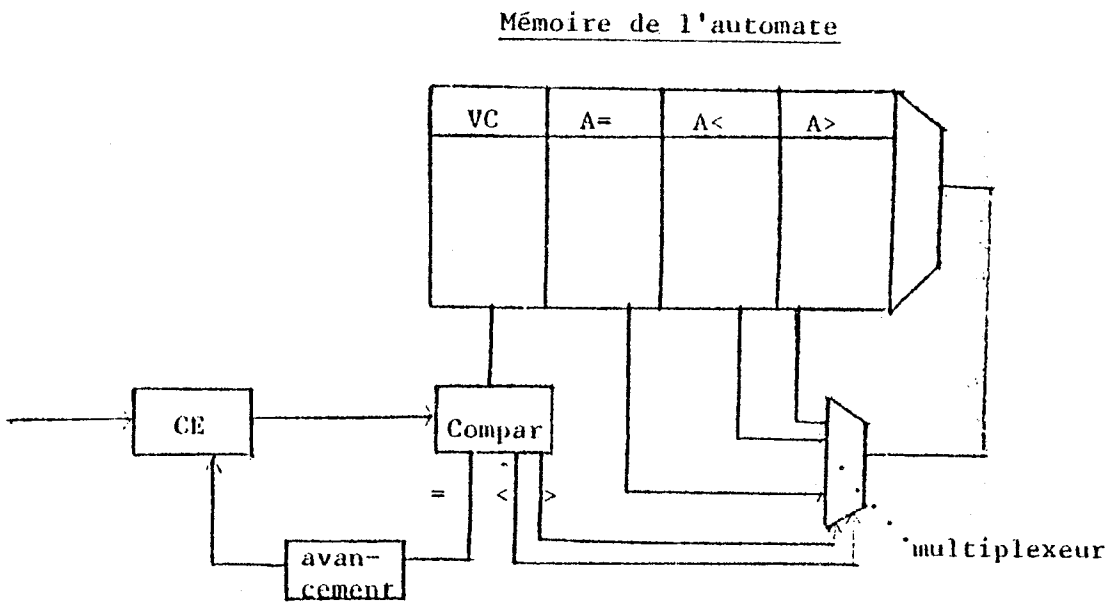
-si le caractère en entrée est supérieur à VC
  * aller à l'adresse A>
  * (ne pas lire le CE suivant en entrée)

L'automate devient:



Les flèches < > n'aboutissant pas ne sont dessinées

Le matériel nécessaire pour interpréter une telle instruction est très simple:



Analysons les performances de cette solution.

a) occupation mémoire

pour chaque caractère, il faut la place du caractère lui-même plus TROIS adresses. On peut minimiser ce facteur de plusieurs manières:

-une des adresses peut être remplacée par un branchement implicite à la position mémoire suivante

-les adresses A< et A> peuvent être exprimées en RELATIF, et limitées à log2 (N) bits, si N est la taille de l'alphabet

Si l'on veut avoir au plus 64K caractères dans l'automate, on peut par conséquent avoir une instruction. tenant sur 32 bits:

- 8 bits pour VC
- 16 bits pour A=
- 8 bits pour A<

N.B. Un automate de 64k caractères peut par exemple reconnaitre plus de 8000 mots de 8 caractères de longueur en moyenne,

Dans des applications de recherche documentaire, les automates dépasseront rarement quelques dizaines de mots, L'adresse A= pourra tenir sur 8 bits, et l'instruction totale sur 24 bits.

b) vitesse d'exécution

Entre deux caractères en entrée, on a besoin d'exécuter au plus log2(N+1) instructions, si N est la taille de l'alphabet. Si l'on désire suivre le débit des disques les plus rapides actuels (un octet toutes les 800 nanosecondes), il suffit donc que notre instruction s'exécute en 100 nanosecondes, ce que est à la portée des technologies standard depuis plusieurs années.

AINSI, ON RESOUT LA QUESTION EN UNE SEULE PASSE

A LA VITESSE DU DISQUE

INDEPENDAMMENT DE LA COMPLEXITE DE LA QUESTION

En fait, la contrainte est beaucoup moins forte, car si l'on avait à exècuter n fois de suite 8 opérations, cela signifierait que , pour n caractères lus en entrèe, on aurait 256 à la puissance n états terminaux dans l'automate! Inversement, pour une taille maximum donnée de l'automate, on peut borner le nombre moyen de comparaisons par caractère lu en entrèe.

Ainsi, sur notre exemple prècèdent, ce nombre moyen vaut 1,17.

(1,17 = somme des nombres d'instructions pour atteindre la fin des mots divisèe par la somme des longueurs de chaque mot.)

Si la mèmoire de l'automate contient au plus 256 instructions et si la longueur moyenne d'un mot clè est de 8 octets, il y a au plus 128mots clès, et il faut en moyenne moins de 7 comparaisons pour choisir l'état terminal de l'automate, soit moins de (7+ 8) ÷ 8 = 1,875 instructions par caractère en entrèe.

Donc les problèmes de vitesse ne sont pas aussi sèvères qu'un examen rapide peut le laisser croire, examen rapide qui a conduit beaucoup d'auteurs à proposer des architectures surdimensionnèes et/ou trop rigides.

Pour profiter de cette constatation au niveau de l'architecture, il suffit de prèvoir une FILE D'ATTENTE des caractères en entrèe. Il existe dans le commerce des boitiers très rapides, vèritables 'producteur-consommateur' câblès, qui rèalisent très bien cette fonction, et permettent donc:

- soit de faire une machine plus lente suivant les besoins MOYENS et non MAXIMAUX

- soit de faire une machine plus rapide qui peut se permettre de perdre un certain temps par endroits (par exemple pour faire des CALCULS auxiliaires), car elle va se rattrapper lorsqu'elle aura simplement à parcourir l' automate.

c) difficultès de compilation

L'algorithme de compilation de l'automate de reconnaissance

d'une liste de mots est conceptuellement assez simple, mais son exécution est longue:

- tri de la liste de mots
- mise sous forme arborescente
- parcours récursif de l'arbre pour générer les instructions

Ses performances sont acceptables pour compiler la question frappée à un terminal par un utilisateur, ne comportant que quelques de mots clés. Elles sont moins acceptables si l'on utilise un thésaurus d'une base de données documentaires ou pour transformer un fichier en automate afin de réaliser l'opération de jonction de l'algèbre relationnelle.

Un dernier point fondamental est que nos instructions peuvent faire les comparaisons < = > et pas uniquement = et ≠. Ainsi, on peut les utiliser non seulement pour savoir si un texte contient un mot donné, mais aussi si il contient (par exemple) un nombre de plusieurs chiffres compris entre tel ou tel intervalle. Plus généralement, on va pouvoir effectuer des recherches faisant intervenir des RELATIONS D'ORDRE, ce qui ouvre énormément de débouchés:

- informatique de gestion et pas seulement informatique documentaire
- traitement des fichiers inverses et des index, et pas seulement des fichiers de données

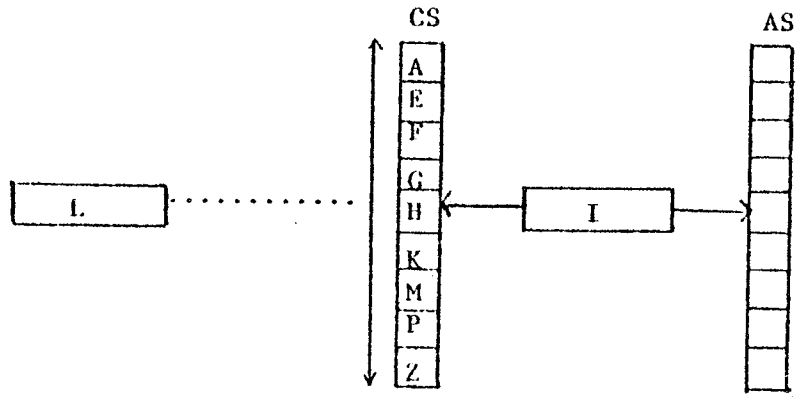
A notre connaissance, aucune autre architecture proposée à ce jour ne permet une telle universalité dans les problèmes traités.

3 3 2 3 Automates dichotomiques interprètés

Plutot que de COMPILER l'arbre dichotomique de reconnaissance d'un caractère parmi N, on peut faire comme habituellement en logiciel, c'est à dire INTERPRETER la recherche dichotomique. Il faut pour celà disposer de:

- la liste ordonnée CS des caractères
- la liste des adresses AS dans le même ordre

- l'adresse d'échec AE
- de deux registres:
  * l'un pointant sur le caractère courant que l'on compare, soit I
  * l'autre contenant la longueur du segment courant dans lequel on fait la comparaison, soit L



L'algorithme est le suivant:

1) au départ, L est égal au nombre de caractères dans la liste et I vaut L÷2 (division entière, donc décalage) à droite)

2) on compare le Iême caractère de la liste avec le caractère courant

3) si ils sont égaux, alors aller à 6

4) si le caractère courant en entrée est inférieur, alors faire en série:
   L ← L ÷ 2

I ← I - L ÷ 2
si L = 0 aller à 7 sinon aller à 2

5) si le caractère courant en entrée est supérieur alors faire en série:
L ← (L - 1) ÷ 2
I ← I + L ÷ 2
si L = 0 aller à 7 sinon aller à 2

6) le caractère a été trouvé en Ième position, l'adresse de l'état suivant est la Ième de la liste d'adresses

7) le caractère n'appartient pas à la liste, l'adresse de l'état suivant est l'adresse d'échec

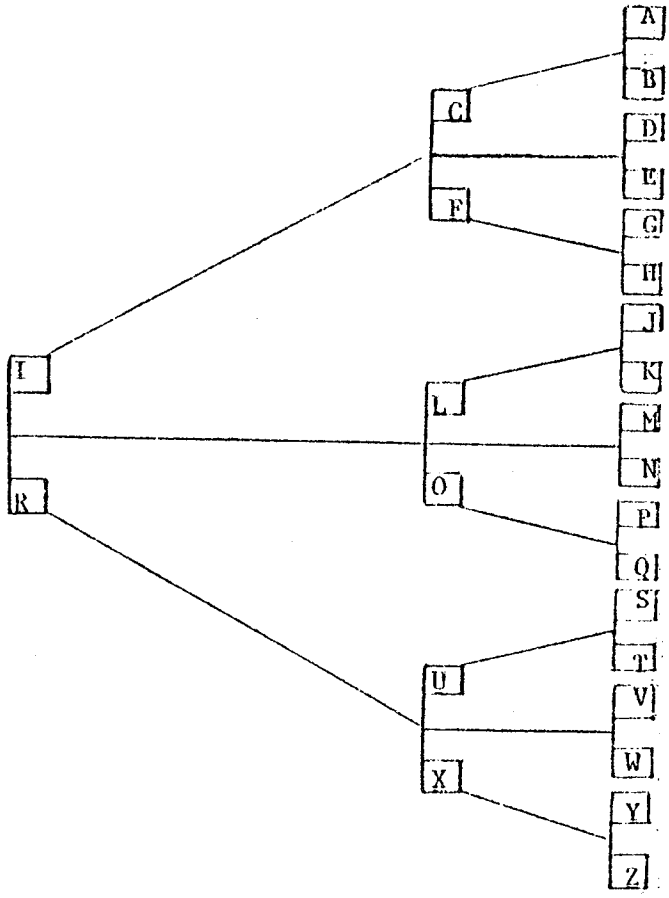
Les calculs nécessaires à cet algorithme peuvent être soit réalisés par des circuits élémentaires, soit être tabulés dans des mémoires mortes de taille raisonnable: 4 fois NlogN bits, si l'alphabet a N éléments.

L'avantage principal de cette méthode est de ne nécessiter que le nombre minimum d'adresses, comme la représentation de Bird. Elle oblige à représenter les caractères, et pas seulement leur table de présence, mais peut travailler sur un gros alphabet, non limité à 8 ou 16 éléments.

3 3 2 4 Une généralisation: utiliser n comparateurs et des automates n- aires

Avec les méthodes précédentes, on traite 2**p - 1 successeurs en p comparaisons maximum.

Si l'on désire accélérer le parcours dans l'automate, une idée triviale est de comparer le caractère en entrée non pas à UN seul caractère, mais EN PARALLELE à PLUSIEURS, TOUT EN CONSERVANT L'EQUILIBRAGE DE L'ARBRE DES COMPARAISONS. Voici l'arbre pour traiter 26 caractères avec DEUX comparateurs en TROIS cycles de comparaison:



Plus généralement, avec n comparateurs, on a les performances suivantes:

Soit P[i] le nombre de caractères traités au iéme cycle de comparaisons, et T[i] le cumul de P[i], c'est à dire le nombre total de caractères traités en i cycles de comparaison.

RAPPEL :

$$\sum_{i=1}^{k} n^{i-1} = \frac{n^k - 1}{n-1} \qquad (1)$$

$P[1] = n$

$P[2] = (n+1) \times n$

$P[i] = \frac{P[i-1]}{n} \times (n+1) \times n$ car

au niveau $i-1$ il y a $\frac{P[i-1]}{n}$ instructions de comparaison en //.

De chacune de ces instructions partent $n+1$ Flèches vers les instructions du niveau suivant, chacune de n caractères.

Donc $P[i] = (n+1)^{i-1} \times n$

$$T[k] = \sum_{i=1}^{k} P[i] = n\sum(n+1)^{i-1} = n\,\frac{(n+1)^k - 1}{n+1-1} \quad \text{(d'après (1))}$$

$T[k] = (n+1)^k - 1$

On a :

Le nombre moyen de comparaisons pour trouver l'état suivant est évidemment inférieur à k et égal à :

$$M[k] = \frac{\sum_{i=1}^{k} i \times P[i]}{T[k]}$$

$$\sum_{i=1}^{k} i \times P[i] = n \sum_{i=1}^{k} i\,(n+1)^{i-1} = \text{(en dérivant (1))}$$

$$= n\,\frac{k\,(n+1)^{k+1} - (k+1)(n+1)^{k} + 1}{(n+1-1)^{2}}$$

$$M[k] = \frac{k\,(n+1)^{k+1} - (k+1)(n+1)^{k} + 1}{n\,((n+1)^{k} - 1)}$$

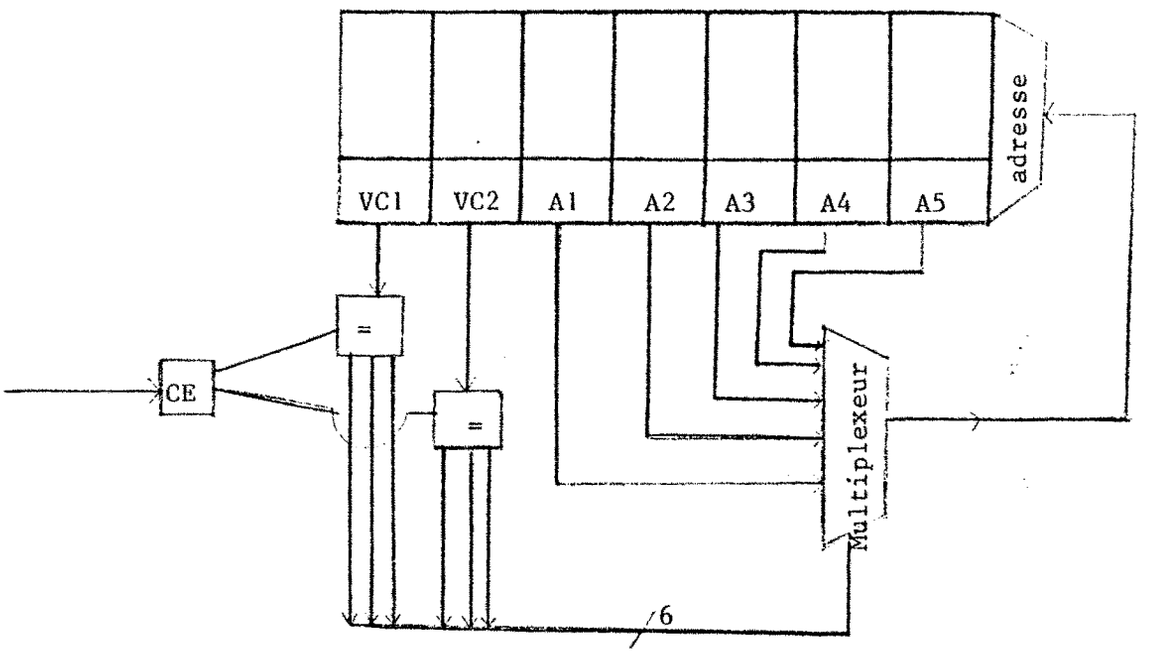
Exemple pour $n = 1$:

$$M[k] = \frac{k\,2^{k}}{2^{k}-1} = k-1 + \frac{k}{2^{k}-1}$$

Le prix à payer pour cette accélération est bien sur que, comme dans TGI [20], plus il y a de comparateurs en parallèle, plus leur taux d'utilisation moyen est mauvais, comme l'est aussi celui de la mémoire: on prévoit toujours n successeurs dans l'automate même si il y en a beaucoup moins. Par contre, on conserve intégralement la souplesse de programmation

La machine interprétant un tel automate aurait l'allure suivante (pour n = 2):

Mémoire de l'automate



En pratique, cette solution peut servir

- soit à utiliser des mèmoires plus lentes: avec 4 comparateurs on traite 24 caractéres en DEUX accés mèmoire.

- soit à suivre des débits très rapides: si en moyenne le nombre de CS est de 10 par état, (ce qui est énorme), il suffit de deux cycles par caractère avec trois comparateurs.

Il devient très possible de suivre des débits de l'ordre de 50 Mhz, même avec de très gros automates, donc des questions très complexes.

3 4 LES AUTOMATES TABULES

3 4 1 INCONVENIENTS ET AVANTAGES

Nous étudions ici des techniques pour tirer le meilleur parti possible de la solution 'naive' exposée en 2 2 2, où le CE est utilisé comme un index dans la liste AS. L'avantage est qu'un seul accès mémoire suffit pour chaque CE, mais l'inconvénient est que AS comporte toujours N éléments, même si il y a très peu de successeurs.

Par exemple, si on se limite aux lettres, chaque état est composé de 26 adresses. Pour représenter UN état, il faut 52 octets, si les adresses sont sur 16 bits. Ceci est évidemment inadmissible.
Avec une tabulation naive:

1 caractère →→ N adresses

Nous avons vu comment la représentation de Bird [26] contournait le problème, en indiçant non pas AS mais la suite de bits constituant la table de présence de AS d'une part, et d'autre part en utilisant un petit alphabet de 8 ou 16 éléments. En conséquence, un caractère habituel est représenté par deux 'petits' caractères, chacun nécessitant une adresse. En négligeant la taille de la table de présence, la représentation de BIRD est telle que:

1 caractère →→ 2 adresses

Notre solution dichotomique interprétée ne testant pas les inégalités est telle que:

1 caractère →→ 1 caractère + 1 adresse

ce qui est meilleur dès que les automates sont grands.

Pour traiter les < et > (dont l'importance est considérable), il faut utiliser une dichotomie avec 2 adresses:

1 caractère →→ 1 caractère + deux adresses

Ces résultats semblent condamner la solution naive

d'indexation, mais trois remarques vont la 'racheter':

a) Première remarque

Dans un automate usuel, seul un petit nombre d'états a plus d'un successeur. Dans [29], on donne la proportion de 10 pour cent. En pratique, cela signifie que les deux ou trois premières lettres des mots-clès suffisent pour les distinguer les uns des autres. (ceci a été mesuré sur de gros systèmes documentaires existants).

Le reste de l'automate est donc constitué de parties LINEAIRES qu'il suffit de représenter comme telles:

1 caractère →→ 1 caractère

Il faut donc prévoir un mécanisme particulier de parcours de ces parties linéaires, indépendant de la technique choisie pour les parties arborescentes, et qui va relativiser dix fois les avantages et inconvénients de ladite technique ...

Notons de plus que, dans la partie arborescente de l'automate, les caractères représentés sont COMMUNS par définition à plusieurs débuts de mots, ce qui diminue encore l'importance relative de la place qu'ils occupent.

b) deuxième remarque

Il est possible de diminuer la taille occupée par un caractère dans l'automate tabulé en le divisant en p 'petits' caractères, le prix à payer étant de multiplier par p le nombre d'accès mémoire.

Exemple: on a vu en 3 2 2 qu'avec des caractères de 1 bit, il suffit d'UNE adresse par caractère, donc de 8 adresses pour un octet, contre 256 si l'on ne divise pas l'octet.

Toutes les combinaisons sont possibles, et, de manière générale, le problème est le suivant:

- soit un caractère de n bits

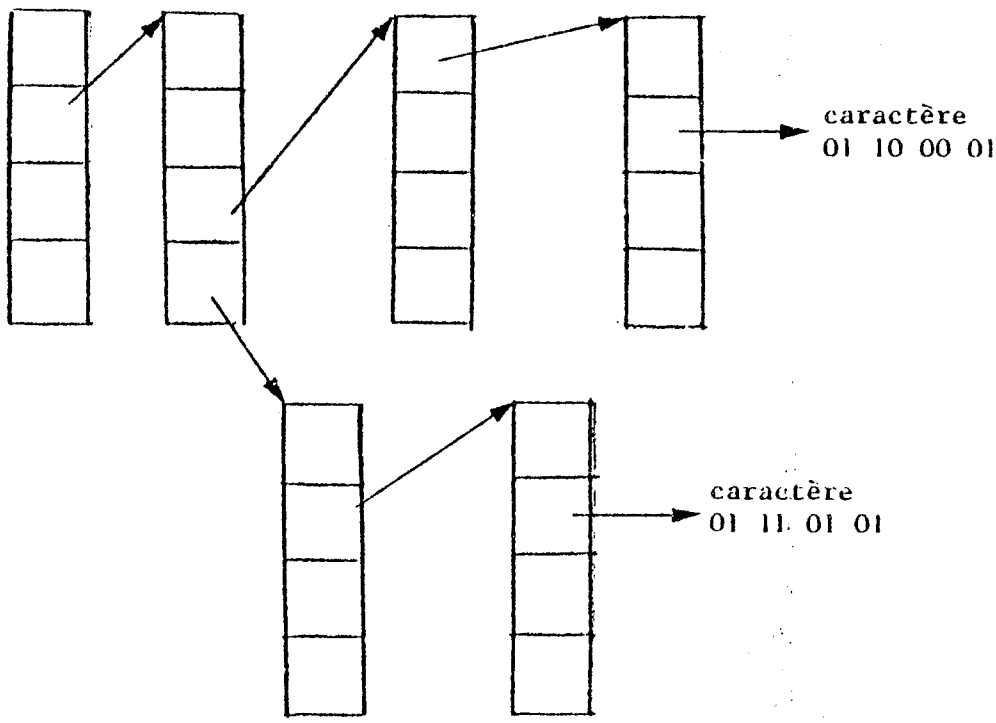
- on peut le décomposer en p caractères successifs, chacun de longueur l1,l2,...,lp

- il faudra p accès mémoire pour traiter un 'gros' caractère

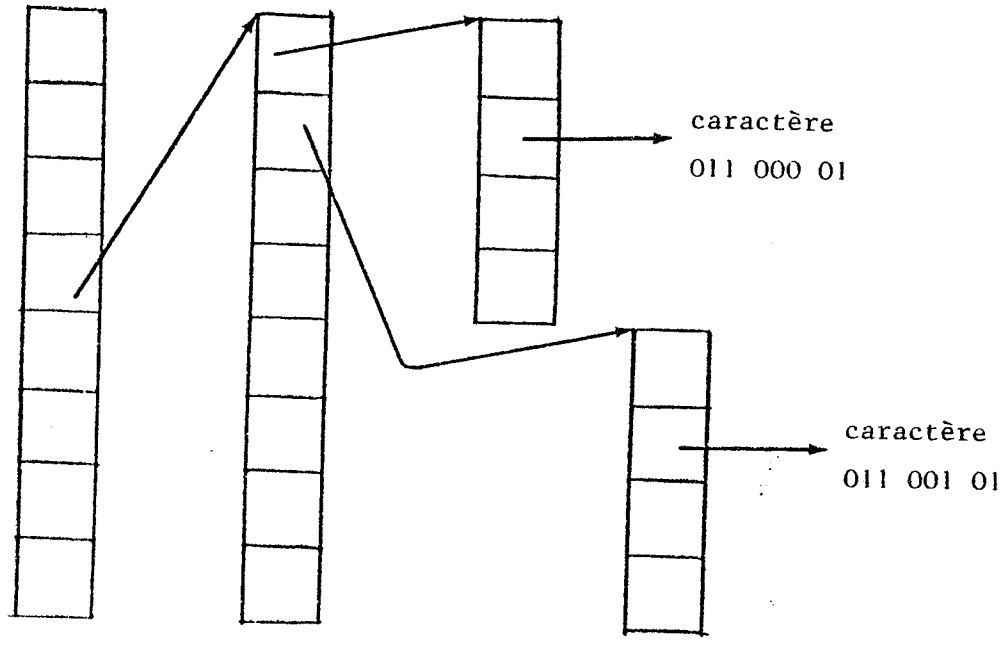
- il faudra 2**l1 + 2**l2 + ... + 2**lp adresses pour le représenter (moins p si une des adresses est implicitement l'adresse courante plus 1)

Exemple:

n = 8, p = 4, li = 2 pour tout i



n = 8, p = 3, l1 = l2 = 3, l3 = 2



Plus gènèralement, si les p petits caractères sont de taille ègale, alors, pour chaque gros caractère:

- la taille occupèe est p (2 ** (n÷p) -1)

- le nombre d'accès mèmoire est p

Toutes ces remarques sont triviales, mais rarement exploitèes dans les architectures proposèes dans la littèrature ou rèellement construites.

c) troisième remarque

Un automate indexè peut être compilè en TEMPS LINEAIRE

L'algorithme détaillé de cette construction est donné en annexe [AI]; il est facile de voir brièvement ici comment les choses se passent.

Supposons que l'on veuille construire l'automate de reconaissance d'un ensemble de mots-clès. Il peut être construit incrèmentiellement de la manière suivante pour chaque nouveau mot:

- avec le caractère courant CE du nouveau mot, on indice la liste AS courante

- si AS[CE] contient une adresse, c'est l'adresse de la liste AS suivante, que l'on indicera avec le caractère suivant, et ainsi de suite.

- si AS[CE] ne contient pas d'adresse, alors, on CREE une nouvelle liste AS' vide, dont on range l'adresse dans AS[CE], et on se ramène au cas précédent.

3 4 2 UNE ARCHITECTURE UTILISANT LES AUTOMATES TABULES

A partir de l'analyse précédente, nous proposons une architecture possible:

3 4 2 1 Structure de données utilisée

* les octets sonts divisés en 4 tranches de 2 bits

* la partie arborescente de l'automate est rangée dans une mémoire appelée AT. Elle contient des adresses vers elle-même et vers l'autre mémoire, CDC. AT peut aussi être vue comme un ensemble de blocs de 4 adresses, et une adresse vers AT est toujours une adresse de bloc, ce qui fait gagner 2 bits pour la coder, et peut permettre un accès parallèle au bloc si les contraintes de vitesse l'exigent.

* la partie linéaire de l'automate est rangée dans une

mémoire appelée CDC. Elle contient des caractères de 8 bits et des adresses vers AT

Dans un premier temps, on supposera qu'un mot de CDC est assez grand pour contenir soit une adresse, soit un caractère de 8 bits, et un bit pour les distinguer. Ce point sera optimisé par la suite.

Dans chaque mot de AT, il faut indiquer si:

- il contient une adresse vers AT
- il contient une adresse vers CDC
- il est vide

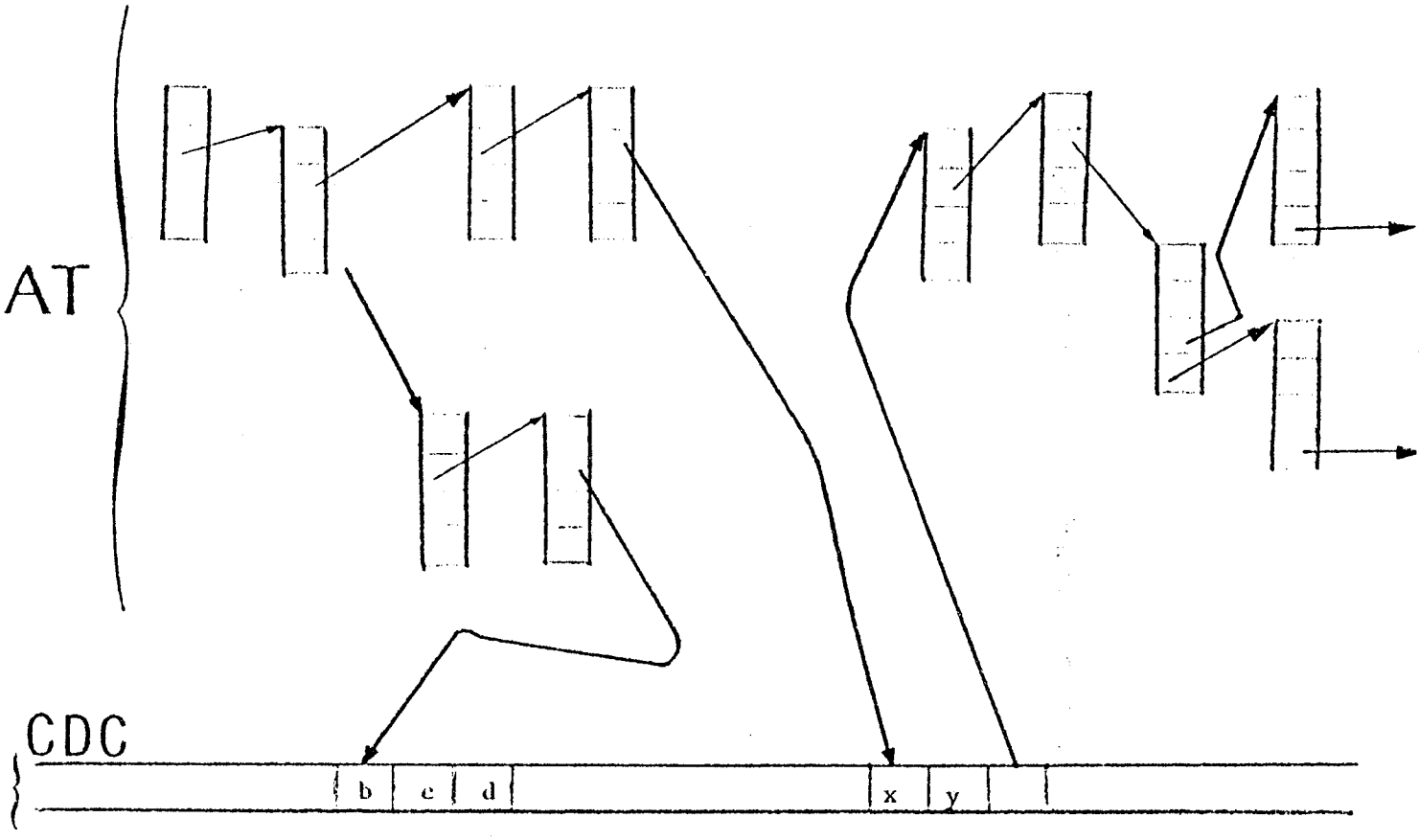
De même chaque mot de CDC indique si

- il contient un caractère et l'unique caractère suivant est situé juste après lui

- il contient un caractère qui a PLUSIEURS caractères suivants, en conséquence de quoi le mot situé juste après est une adresse vers AT

Exemple de la représentation de l'automate reconnaissant les mots

a b c d	avec	a ≡ 01 10 01 01
e x y z	avec	z ≡ 01 01 10 11
e x y t	avec	t ≡ 01 01 11 11
	avec	e ≡ 01 01 10 01



3 4 2 2 L'algorithme de crèation de l'automate et son implèmentation

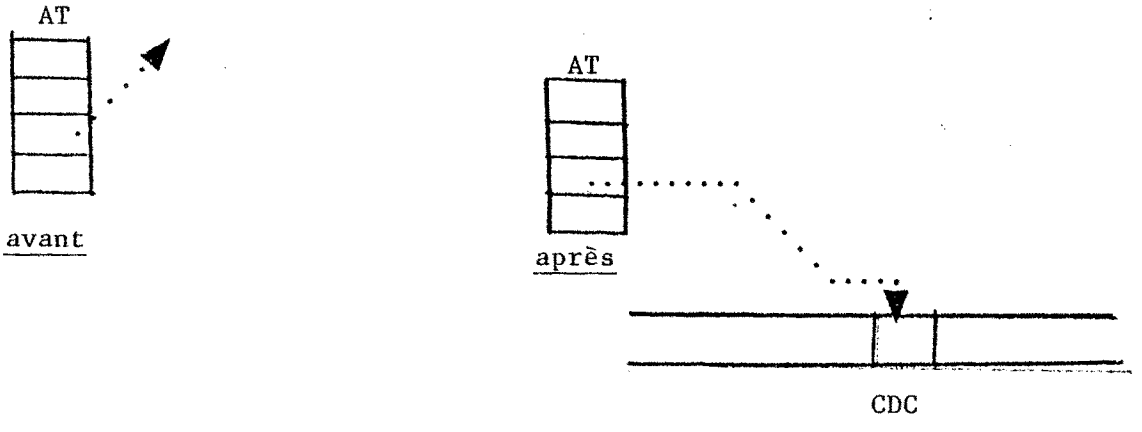
Comme le principal intèrèt de cette reprèsentation est qu'elle peut être crèée en temps linèaire, nous commen?ons par ètudier sa crèation avant son utilisation.
Le principe est le suivant:

L'algorithme re?oit une suite de mots sèparès par des caractères spèciaux. Quand on stocke un mot, on stocke avec lui son dèlimiteur de fin, ceci afin de distinguer entre deux mots dont l'un serait le dèbut de l'autre:
PARIS/ et PARISIEN/
Chacun de ces mots correspondra à une feuille diffèrente dans l'arbre.

L'algorithme crée l'automate caractère par caractère, et peut prendre 6 états:

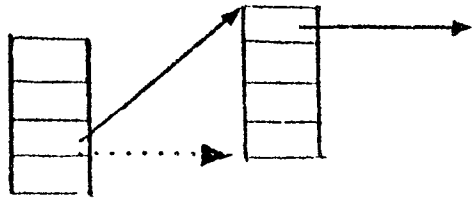
- au début d'un mot, ETAT ← 2

- ETAT = 1 : le caractère précédent était rangé dans AT, et il ne partageait pas sa case avec un autre. Donc la partie arborescente est terminée, et le caractère courant est rangé dans CDC, et ETAT suivant ← 4

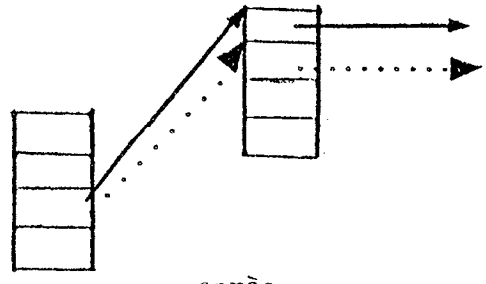


- ETAT = 2: le caractère précédent était rangé dans AT, et il partageait sa branche avec un caractère égal d'un autre mot continuant en AT. Il faut donc placer le caractère courant dans AT, en continuant la même branche

N.B. Les flèches pleines correspondent à un mot déjà présent et les flèches pointillées au mot en cours de rangement.

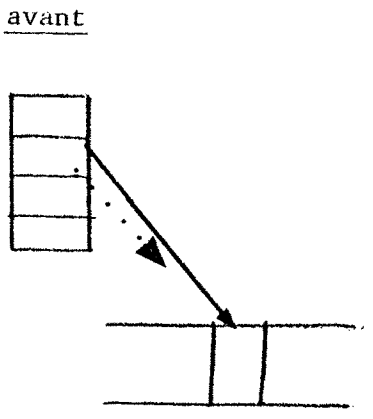


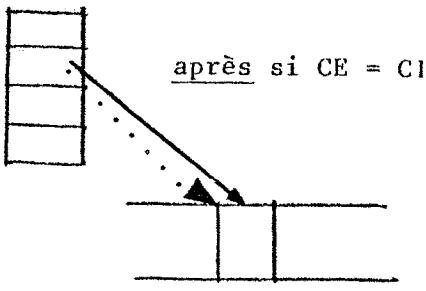
avant

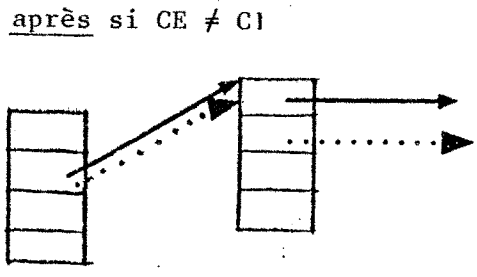


après

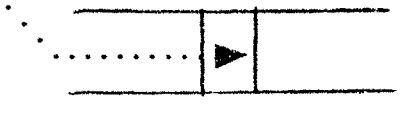
- ETAT = 3: le caractère précédent était rangé dans AT, et il partageait sa branche avec un caractère égal d'un autre mot continuant dans CDC par un caractère de valeur C1. Si le nouveau caractère à placer est égal à C1, alors il se confond avec lui, et ETAT ← 5. Sinon, il faut retourner dans AT pour distinguer entre ces deux caractères par une arborescence.

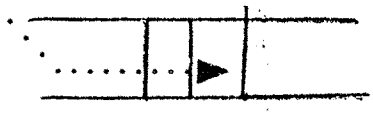




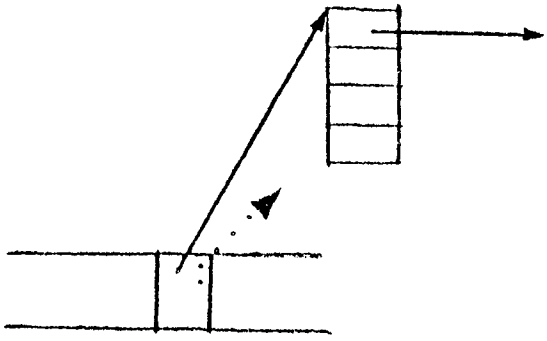


ETAT = 4 : le caractère précédent était seul dans CDC, on range le courant à sa suite; ETAT suivant ← 4

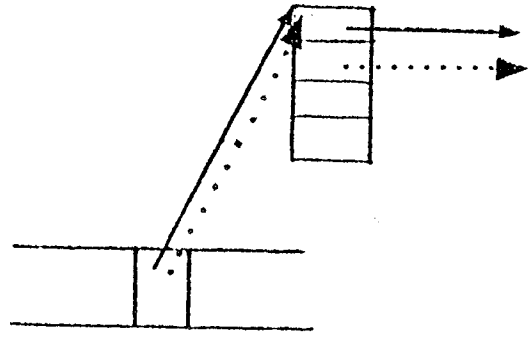




ETAT = 5: le précédent était dans CDC, avec un caractère égal d'un autre mot qui se poursuivait dans AT. Il faut aller placer le caractère courant dans AT.

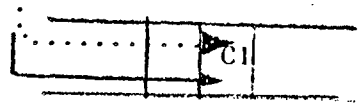


avant



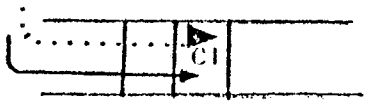
après

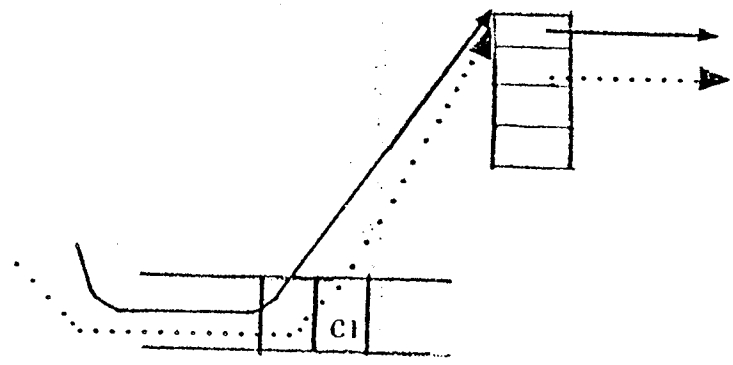
ETAT = 6 : le caractère précédent était dans CDC, avec un caractère égal d'un autre mot qui continuait dans CDC par un caractère C1. Comme pour l'état 3, si le nouveau caractère vaut C1, il se confond avec lui; sinon il faut continuer en AT pour construire un arbre qui va distinguer entre ces deux caractères.



après si CE = C1

avant





après si CE ≠ C1

N.B. L'ETAT suivant est soit 1, 2, ou 3 lorsque ce n'est pas spécifié dans ce court résumé de l'algorithme.

3 4 2 3 Performances de cet algorithme

Cet algorithme a été envisagé car il avait intuitivement de bonnes qualités, outre sa constructibilité en temps linéaire:

- la division d'un octet en 4 tranches de 2 bits conduit à 4 accès mémoire maximum par octet, et 16 adresses maximum par octet

- sa complexité n'est pas excessive

- il fait un grand usage de mémoire vive, avec une longueur de mots faible, puisque ces mots contiennent des adresses plus 1 ou 2 bits. Par exemple, une taille de mots de 16 bits permet de représenter des automates de plusieurs dizaines milliers de caractères.

Il restait à confirmer ces impressions intuitives. Pour cela, nous avons simulé et mesuré le comportement de l'algorithme à l'aide d'un programme écrit en APL. (cf annexe A1). Le jeu d'essai retenu a consisté à construire l'automate de reconnaissance des prénoms du calendrier. Les caractéristiques de cet ensemble sont les suivantes:

- 325 mots

- 2487 caractères, y compris le délimiteur de fin de chaque mot

Les performances suivantes ont été obtenues:

* En vitesse:

- 6286 cycles d'accès mémoire, lecture et écriture confondues, les acces à AT et CDC étant faits en parallèle quand cela a un intèret, ce qui se présente très rarement d'ailleurs.

Donc, il y a eu 2,53 ACCES MEMOIRE PAR CARACTERE EN ENTREE EN MOYENNE.

Ceci est un résultat très satisfaisant, qui permet d'envisager l'utilisation de mémoires MOS dynamiques bon marché pour représenter de gros automates. Si il s'agit de suivre un débit de disque à 10Mhz, il suffit dans notre cas d'un accès toutes les 316 nanosecondes en moyenne. Ceci confirme encore l'intérêt d'une file d'attente câblée des caractères en entrée, car, dans certains cas, il faut faire plus de 8 accès mémoire par caractère (cas des ETATS 3 et 6), ce qui imposerait un cycle inférieur à 100 ns dans une machine synchrone.

* En taille mémoire:

- 3533 mots dans AT, 1401 mots dans CDC

Si on suppose que les mots dans CDC sont aussi longs que ceux de AT, (ce qui n'est pas optimal et peut être évité comme expliqué plus bas), on arrive à

1,98 MOTS par caractère.

Si les mots sont de 16 bits, il faut donc 4 octets pour en représenter un seul. Le facteur d'expansion est de 4. En fait ici, 12 bits suffiraient pour adresser la mémoire utile, donc le facteur d'expansion peut descendre jusqu'à 3.

Le problème de la taille des mots de CDC est que ces mots contiennent soit:

- une adresse vers AT qui peut être grande

- un caractère plus un bit indicateur

et le deuxième cas est le plus fréquent (93 pour cent des cas).

Il serait plus intéressant que chaque mot contienne un octet, et que les adresses soient sur deux mots consécutifs. Le gain en mémoire sur CDC (sans parler des pertes sur AT, très rares),

permet le facteur d'expansion suivant:

- 9 bits pour CDC

- 12 bits pour AT

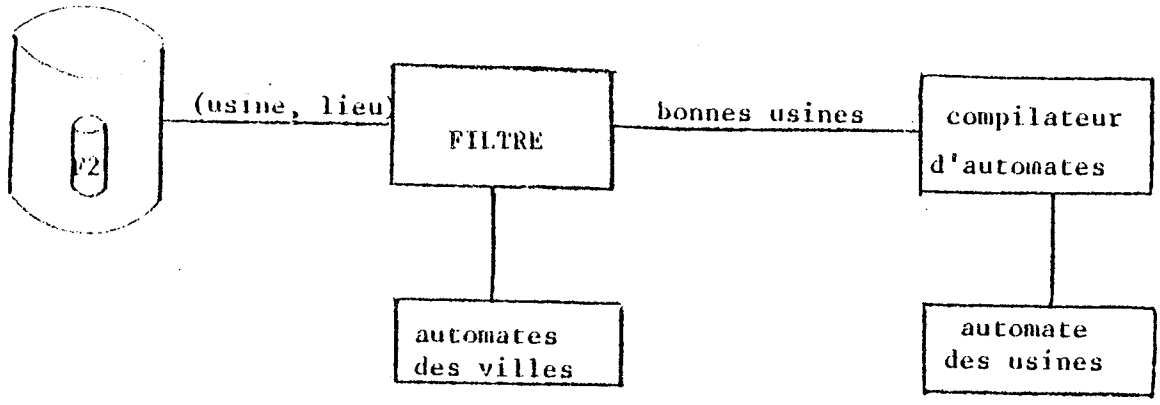
Soit (3533×12) + (1401×1.07×9) ↠ 2,80 soit un progrès de 7 pour cent par rapport à la solution non optimisée.

3 4 2 4 JONCTIONS AU VOL

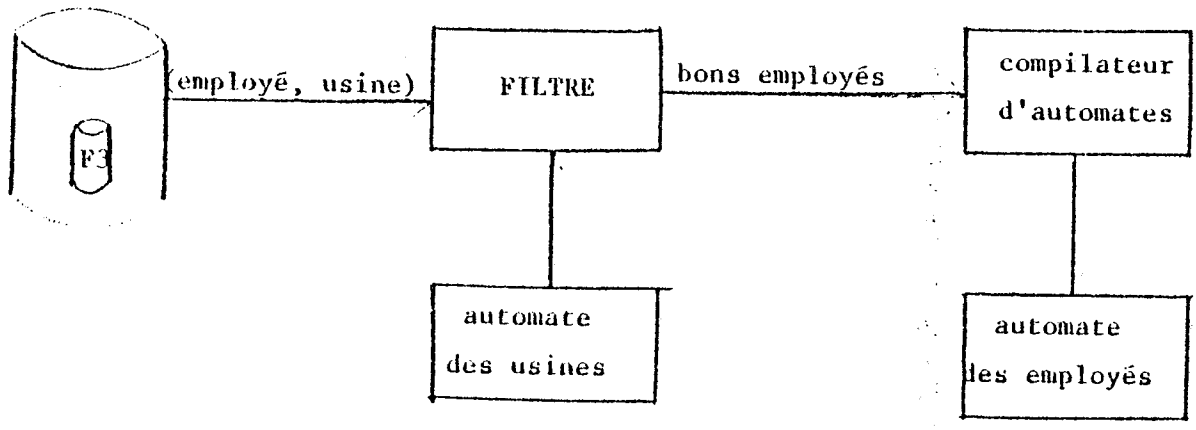
En conclusion, ces performances nous satisfont beaucoup car elles permettent d'envisager avec rèalisme la possibilitè d'exècuter l'opèration de JONCTION entre deux relations non seulement en temps LINEAIRE, mais encore mieux: AU VOL.

Exemple:
Soient deux fichiers F1: (employè, usine), et F2: (usine, lieu)
On veut la liste des employès dont l'usine est à PARIS ou LILLE ou MARSEILLE.

Dans un premier temps, l'automate qui reconnait ces trois villes est construit, et on explore F2 avec un filtre contenant cet automate. Chaque fois qu'il trouve un enregistrement avec l'une de ces trois villes, le filtre envoie le champ usine vers le compilateur d'automates, qui travaille donc en pipeline avec le filtre. A la fin de l'exploration de F2, il suffit de balayer F1 avec le filtre utilisant l'automate que l'on vient de construire.

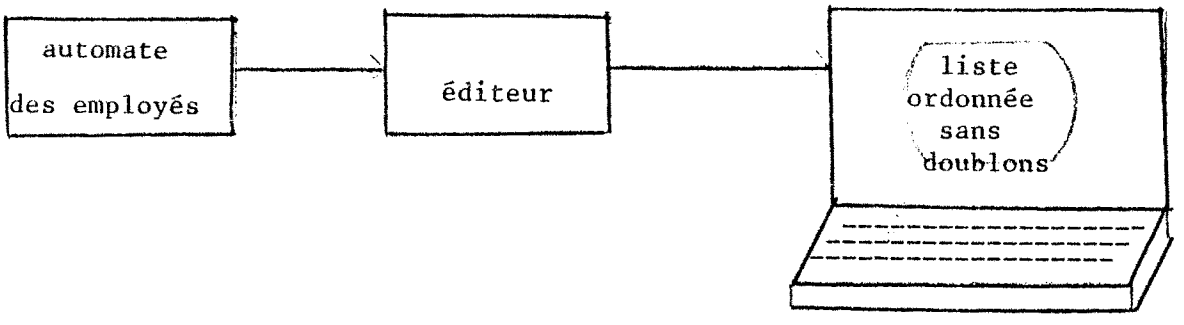


Au cours de ce second balayage, le filtre sélectionne les enregistrements (employé,usine) dont l'usine est reconnue par l'automate construit au vol. Il envoie vers le constructeur d'automate le champ employé de ces enregistrements, ce qui a pour effet d'éliminer les noms d'employés figurant plus d'une fois (opération dite de PROJECTION en algèbre relationnelle).



On dispose finalement de l'ensemble des noms d'employés recherchés sous la forme de l'automate qui les reconnait. Un

algorithme très simple et facile à réaliser de manière rapide peut alors restituer cet ensemble DANS L'ORDRE ALPHABETIQUE, et toujours dans un temps linéaire.



En conclusion, l'opération:

''Donner la liste ordonnée des employés travaillant dans une usine située à PARIS ou LILLE ou MARSEILLE''

a été réalisée dans un temps que l'on peut qualifier d'OPTIMAL au sens fort où il est égal au temps nécessaire pour LIRE LES OPERANDES DE CETTE INSTRUCTION.

A notre connaissance, aucune autre architecture n'a été construite ou proposée avec les mêmes performances.
Le prix principal à payer est la taille mémoire pour représenter l'automate. La base de cette technique est en effet de TRANSFORMER UN FICHIER EN QUESTION, en un automate plus précisément, et de mettre cet automate en mémoire aléatoire.

Ce dernier point peut paraitre aberrant: 'si le fichier est en mémoire, alors il n'y a évidemment plus de problème!'

A cela nous répondons que:

- le fichier en mémoire est un fichier intermédiaire, résultant lui-même en général d'une opération de sélection. Il n'est pas nécessairement énorme en pratique.

- si il est néanmoins trop gros pour tenir dans la mémoire des automates, il suffit de faire l'opération en plusieurs fois. Le temps de réponse sera au pire proportionnel à la taille de la mémoire centrale, ce qui n'est pas une mauvaise performance en soi.

- puisque c'est la taille de mémoire centrale qui est en question, il ne faut pas oublier que les gros ordinateurs utilisant de grosses bases de données ont souvent 5 à 10 millions d'octets de mémoire centrale. Avec une taille équivalente et un facteur d'expansion de 5 de la taille de l'automate, notre technique accepte des relations intermédiaires de 1 à 2 millions de caractères, soit par exemple de 50 à 100 000 noms propres, ce qui n'est pas si courant ...

- cette taille mémoire peut être diminuée en utilisant une technique de 'hash-coding', au prix d'une augmentation raisonnable du nombre d'accès, comme expliqué à la fin de ce chapitre.

3 4 2 5 Implémentation de l'algorithme

L'analyse du programme en APL donnant le détail de l'algorithme fait apparaître les besoins suivants:

- 1 mémoire AT de mots de largeur log2 P bits, si P est la taille de AT

- 1 mémoire CDC de mots de Q + 1 bits, si Q est la longueur en bits d'un caractère

- 14 registres , la plupart de log2 P bits, dont 5 incrémentables

- 20 liaisons possibles entre ces registres

- 1 comparateur de caractères, 1 multiplexeur, 1 additionneur

Les contraintes du séquencement sont telles que entre deux accès à AT ou CDC on a seulement à faire:
- un calcul combinatoire consistant au plus en une addition, un multiplexage et une comparaison en série.
- puis plusieurs transferts registre à registre pouvant s'effectuer en parallèle si les liaisons correspondantes existent.

Notre but ici n'est pas de proposer un dessin détaillé d'un tel opérateur, mais seulement d'en montrer la faisabilité. Les différentes solutions pour réaliser le séquencement ne sont donc pas étudiées car elles dépendent étroitement des contraintes économiques et de performance que seule une étude de marché peut fixer, ce qui n'entre pas dans le cadre de cette thèse.
Il est néanmoins clair qu'une version en circuits SSI/MSI comporterait moins de 50 boitiers, (sans les mémoires) , et qu'une intégration sur un seul boitier semble possible.

Avant d'entreprendre l'une ou l'autre de ces réalisations, un gros travail de mesures par simulation et aussi de modélisation mathématique doit encore être fait. De plus, des sophistications sont certainement possibles concernant les différents paramétres de l'architecture (taille des mots, découpage des caractéres) et aussi la prise en compte des aspects syntaxiques et sémantiques qui font l'objet de paragraphes ultérieurs.
Le paragraphe qui suit présente une optimisation très intéressante pour le cas des données textuelles.

3 4 2 6 Une représentation des parties arborescentes adaptée aux données textuelles.

Si les données traitées ne sont pas de nature informatique (nombres, clés), mais sont des mots d'une langue naturelle, elles présentent une propriété intéressante: tous les caractéres n'ont pas la même fréquence d'apparition. Voici des chiffres extraits d'une étude [28] sur la langue française, obtenus à partir de l'analyse de textes administratifs.

Pourcentage d'apparition des lettres de l'alphabet:

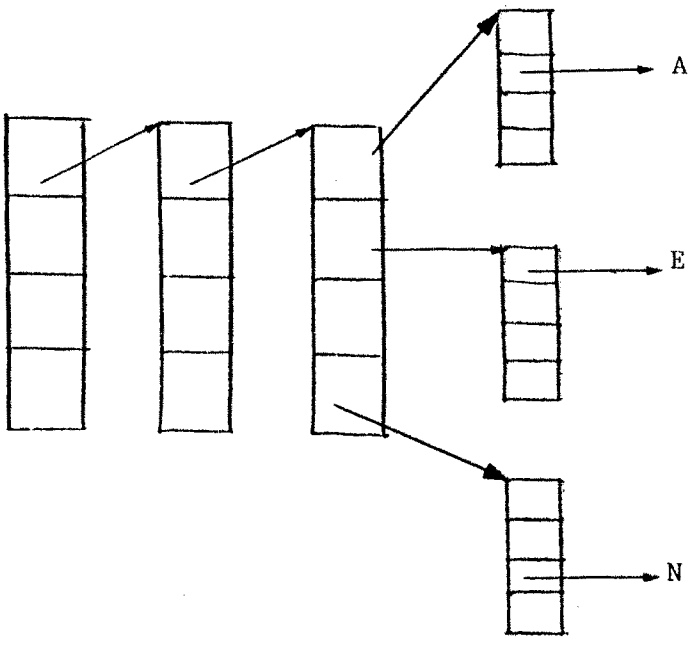
blanc (séparateur de mots)	17,32						
E	14,28	A	6,90	N	6,50	S	6,24
I	6,14	R	5,82	T	5,80	L	4,67
O	4,64	U	4,61	D	3,39	C	2,85
P	2,61	M	2,22	V	1,04	Q	1,04
G	0,98	F	0,95	B	0,68	H	0,48
X	0,35	J	0,24	Y	0,18	K	0,06
Z	0,06	W	0,05				

On constate que les 6 caractères les plus fréquents couvrent environ 50 pour cent des cas, et que les 16 plus fréquents atteignent 94 pour cent. Pour profiter de cette concentration des fréquences, il suffit de coder les caractères dans l'ordre de leur fréquence. C'est ce que nous avions fait dans l'exemple des prénoms précédent:

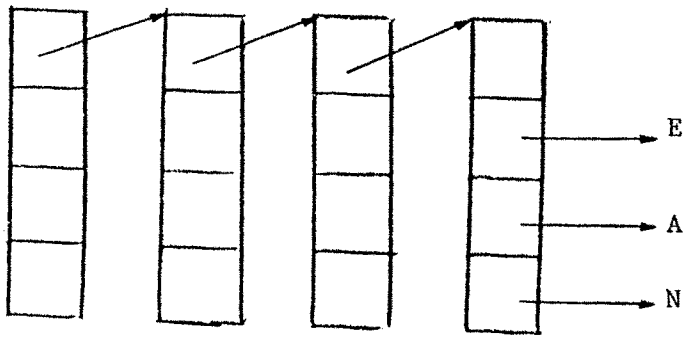
blanc → 0
E → 1
A → 2 etc. ...

Dans ces conditions, l'arbre dans AT est plus étroit et prend moins de place que si le codage est quelconque, suivant l'ordre alphabétique par exemple. Ainsi, pour les trois lettres E , A , N :

* dans l'ordre alphabétique A → 1 , E → 5 , N → 14
D'où l'arbre suivant:



* dans l'ordre des fréquences A → 2 , E → 1 , N → 3
D'où l'arbre suivant:

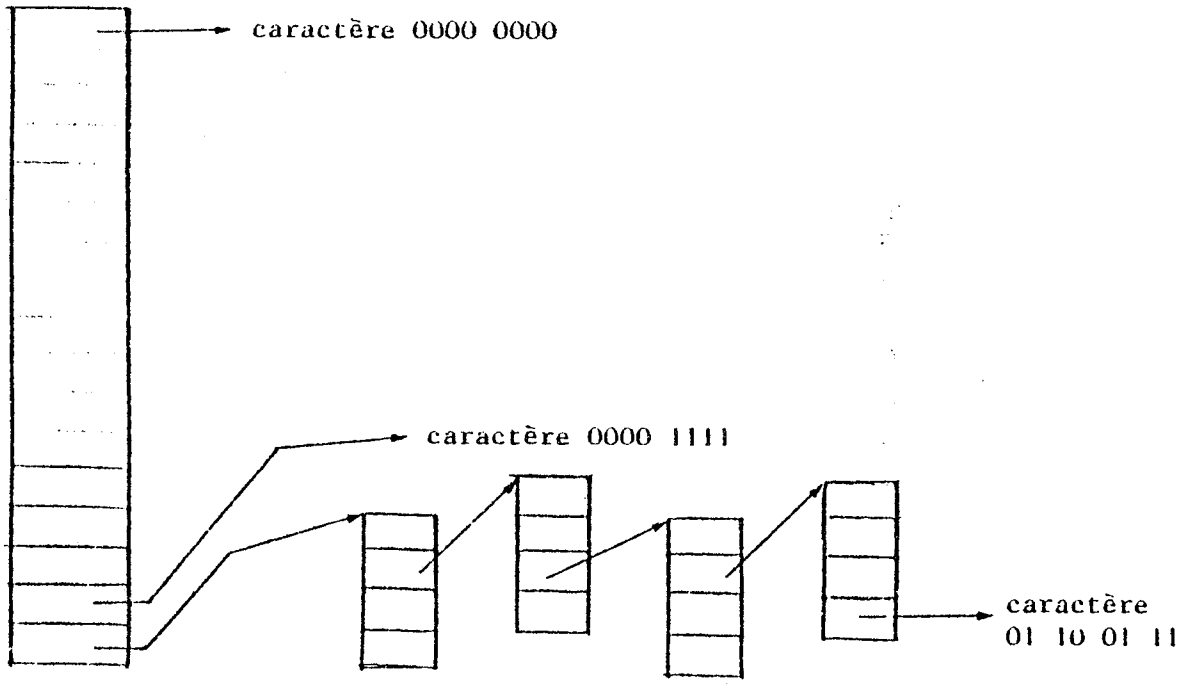


Pour l'exemple des prénoms, avec le codage optimisé, il faut 3533 mots dans AT. Les mesures ont montré qu'il en faut 3807 avec le codage alphabétique. La perte est donc de 7 pour cent.

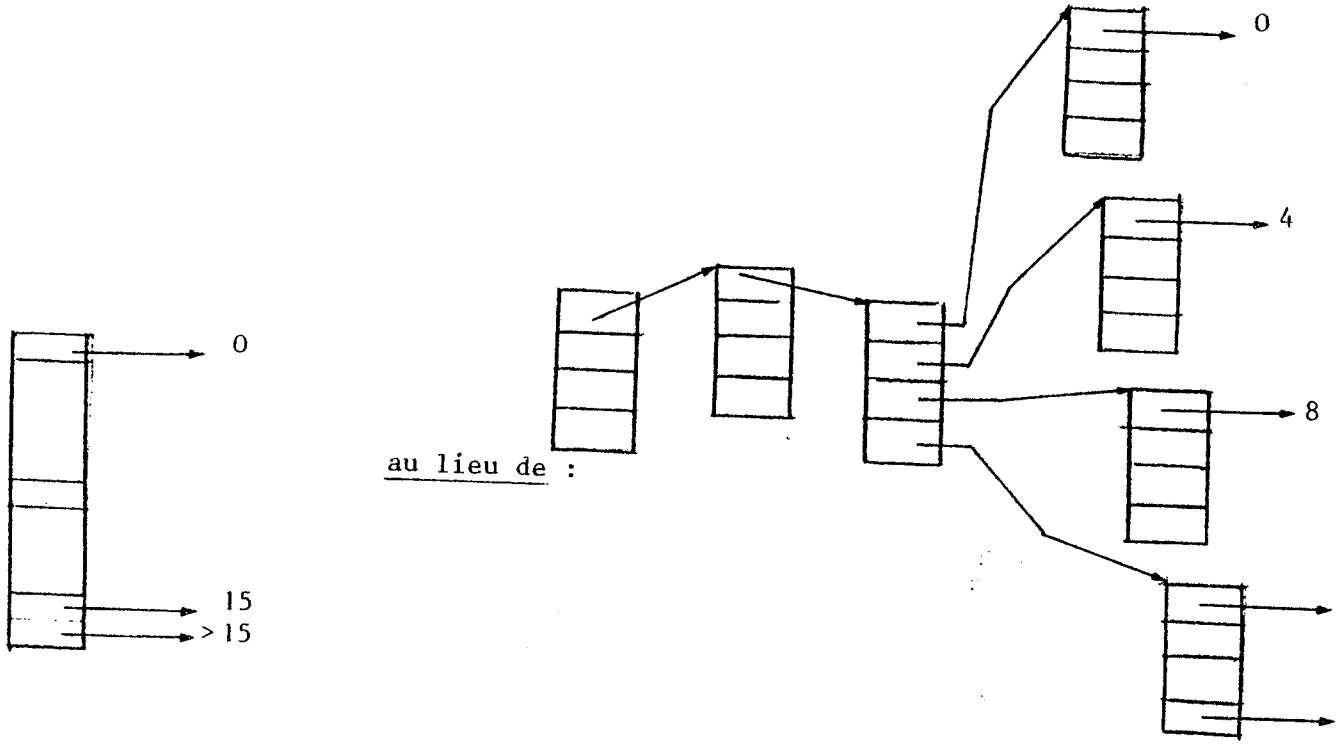
Il y a une manière plus radicale de tenir compte des

fréquences. C'est de revenir à l'indexation 'naive', sans découpage de l'octet, mais seulement pour les n caractères les plus fréquents. Par exemple, pour n = 16, on utilise la structure de données suivante:

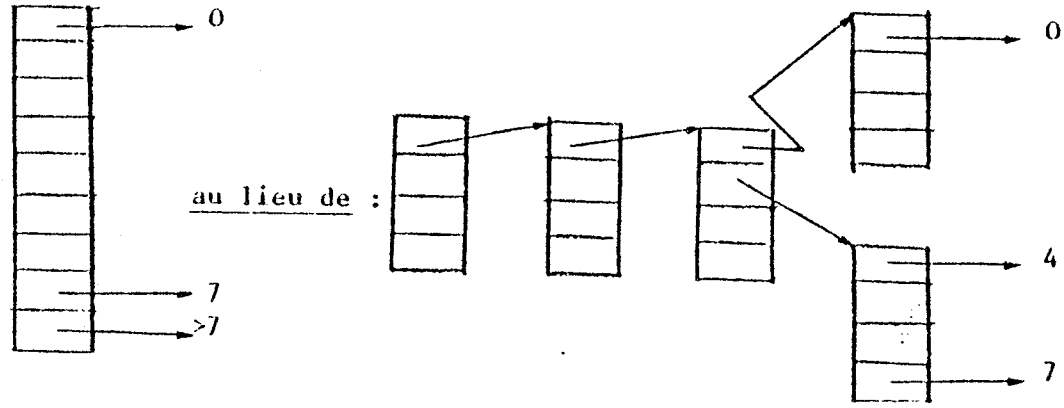
Si le caractère a un code ≤ 16, on trouve directement l'adresse suivante dans la première table, sinon on fait comme de coutume. Exemple:



Ainsi, pour représenter les 16 premiers caractères, on a besoin de seulement 17 adresses au lieu de 28:



Si n = 8, on utilise seulement 9 adresses pour les 8 premiers caractères au lieu de 20:



En plus du gain en taille mémoire, on fait une économie énorme en nombre d'accès mémoire, puisque pour les n premiers caractères, UN SEUL accès est nécessaire, au lieu de QUATRE. Et ceci dans 94 pour cent des cas si n vaut 16!

Ce gain considérable a lieu aussi bien lors de la construction de l'automate que lors de son utilisation.

Avec n = 16, les performances constatées sur l'exemple des prénoms deviennent:

3565 accès mémoire, ce qui ramène à 1,43 accès par caractère, chiffre extrêmement satisfaisant

3725 mots dans AT, ce qui met le taux d'expansion de l'automate à 4,12 avec des mots de 16 bits et à 3,09 avec des mots de 12 bits.

Avec n = 8, il faut 1,8 accès mémoire et le taux d'expansion est de 3,98 avec 16 bits, et 2,98 avec 12.

Si l'on prend n = 27 (toutes les lettres sont tabulées), le nombre d'accès est de 1,22 . Il n'est pas égal à 1 à cause des opérations compliquées faites dans les états 3 et 6.
Ce qui est plus intéressant, est que le taux d'expansion de l'automate n'est pas catastrophique, puisque égal à 5 avec des adresses de 16 bits. Une autre remarque est que le tableau AT ne contient alors que 9.6 pour cent d'éléments non nuls, contre 33 pour cent quand n = 0. AT est plus gros, mais plus vide. Par exemple, dans notre cas, il y a moins de 500 mots de AT non nuls.

3 4 2 7 Compactage de la mémoire AT par hash-coding.

On peut alors essayer de profiter de cette faible densité d'informations utiles dans AT pour le COMPACTER, quitte à perdre sur le nombre de cycles mémoire.
Une solution est de ne mémoriser que les informations utiles de AT, sous la forme d'un ensemble de couples (adresse, valeur). Cet ensemble de couples est organisé par hash-coding sur les adresses.

Pour accéder au mot de AT d'adresse 'a', il faut calculer le hcode de 'a' et explorer la liste de couples correspondante.

Les mesures effectuées sur l'exemple des prénoms aboutissent dans ce cas à:

- un taux d'expansion mémoire de 1.75

- un nombre d'accès par caractère de 2,25 .

Cette solution est celle qui semble réaliser le meilleur compromis temps/mémoire pour la représentation de notre exemple des prénoms. Elle est très intéressante pour permettre la réalisation au vol d'opérations de jonction sur de grandes relations.

3 4 2 8 Calcul d'une borne du taux d'expansion mémoire des automates tabulés compactés.

Soit un ensemble de M mots, chacun de p caractères appartenant à un alphabet de N éléments. Le nombre maximum de mots de cet ensemble est N**p.

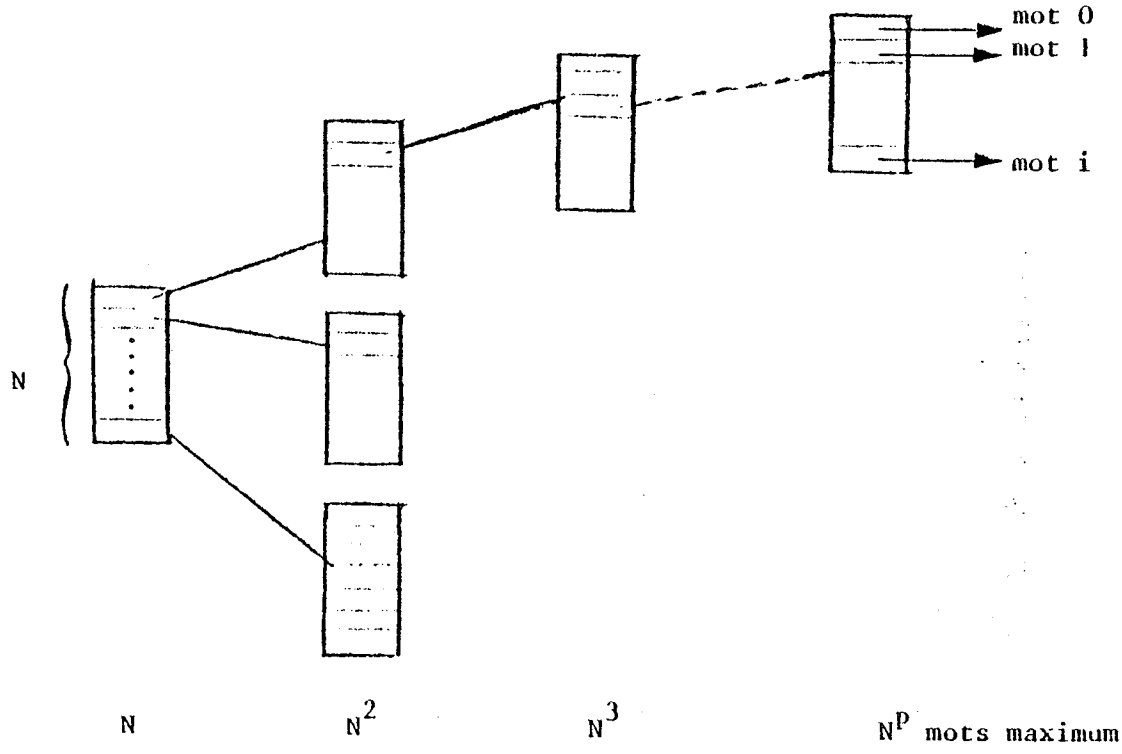
Posons r = (N**p)÷M

Soit T la taille en bits de l'automate tabulé compacté par hash-code représentant ces M mots.

Nous cherchons une borne au taux d'expansion de cet automate, soit au rapport entre T et la taille en bits de l'ensemble des M mots:

TM = p×M×Log N bits

L'automate tabulé a l'allure suivante:



A chaque mot correspond un chemin racine-feuille, tous de longueur p. Le long de ce chemin se trouvent p adresses
Chaque noeud de l'arbre est une table de N adresses. Dans la représentation compactée, seules les adresses effectivement utilisées sont représentées par un triplet dans une liste. Forme d'un triplet:

- adresse du triplet suivant dans la liste

- adresse du mot de AT représenté

- valeur de ce mot

Il faut de plus compter la taille de la table de début des listes de hash-code. Si l'on fait l'hypothèse très satisfaisante pour le temps d'accès que ces listes ne comportent qu'un seul élément en moyenne, il faut donc compter une adresse supplémentaire pour chaque triplet.

Finalement, toute adresse utilisée dans AT occupe au plus 4 adresses dans la forme compactée.
Chacune de ces adresses doit représenter des nombres inférieurs au total de mots possibles de AT, soit (1 -N**(p+1)) ÷ (1 - N); elle sont donc codées sur (p+1) Log N bits au maximum.

Une adresse non vide dans AT occupe donc moins de 4 × (p+1) Log N bits.

Majorons le nombre de ces adresses.

a)- au niveau des feuilles, il y en a (N**p)÷r

b)- aux niveaux inférieurs, il y en a au plus (1 - N**p) ÷(1 - N).

La taille de l'automate compacté TA est bornée en ajoutant les quantités de a) et b), multipliées par la taille en bits précédente.

Après simplification, la division de ce résultat par la taille TM de la représentation ''normale'' des mots donne comme borne du taux d'expansion:

$$4 \times (1 + 1 \div p) \times (1 + r \div (N - 1))$$

Rappelons que r est le rapport entre le nombre maximum de mots (donc de feuilles) et le nombre rèel.

Or, dans la partie arborescente de l'automate considèrèe ici, tout ètat a au moins deux successeurs, sinon il serait dans la partie linèaire de l'automate. Donc r est majorè par N÷2.

Finalement, le taux d'expansion est majorè par:

$$4 \times (1 + 1 \div p) \times (1,5 + 1 \div 2 \times (N - 1))$$

quantitè qui est très faiblement dèpendante de N et p dès que ces paramètres deviennent grands.

Par exemple, pour p = 8 et N = 26, cette borne vaut 6,84

N.B. Le but de ce calcul n'ètait pas de minimiser la borne, mais simplement de prouver son existence et de prouver qu'il n'y a pas de risques d'explosion de l'automate.

Rappelons que, sur l'exemple des prènoms, le taux effectif d'expansion obtenu a ètè de 1,75.

Une étude rigoureuse des performances de la représentation tabulée des automates reste à faire. Notre but a seulement été de montrer leur viabilité sur un exemple non trivial. Ces performances dépendent probablement beaucoup de l'ensemble des mots-clés considérés: distribution de la fréquence des caractères, longueur des mots, noms propres ou communs, mots techniques ou courants. En pratique, du point de vue architectural, ceci conduit à prévoir la possibilité de PARAMETRER dynamiquement les valeurs critiques:

- la valeur de n (nombre de caractères traités en un seul accès mémoire)

- le découpage d'un octet en plusieurs tranches.

Pour chaque champ de chaque fichier, il suffirait d'indiquer les valeurs 'optimales' de ces paramètres, qui pourraient être obtenues expérimentalement.

3 5 UNE AUTRE APPROCHE DU FILTRAGE: LE PREFILTRAGE ELEMENTAIRE MAIS RAPIDE

Plaçons nous dans le cas où le critère de filtrage ne fait intervenir que des EGALITES entre les données analysées et un ensemble de valeurs, à l'exclusion des comparaisons < et >. Il s'agit d'une situation très courante, d'abord en informatique documentaire, mais aussi avec des bases de données classiques lorsque l'on cherche l'égalité entre des constantes et la valeur d'un champ, soit que l'on cherche à repérer des caractères DELIMITEURS de champs, pour savoir si le champ qui suit nous intéresse ou non.

L'idée est de faire précéder le filtrage au niveau d'un enregistrement par un préfiltrage au niveau des constituants de l'enregistrement: mots, champs.

On ne veut envoyer vers le 'vrai' filtre que les constituants de l'enregistrement qui l'intéressent, dans le but de REDUIRE le plus possible son débit d'entrée. Ceci n'aura de sens que si le préfiltre est soit plus simple soit plus rapide (ou les deux à la fois si possible) que le filtre.

Un enregistrement est supposé être découpé en champs. Chaque champ commence par un caractère délimiteur réservé qui lui est propre. A l'intérieur d'un champ, on trouve une suite de mots, un mot étant une suite maximale de caractères appartenant à un ensemble donné (les caractères alphanumériques par exemple).

Le préfiltrage va consister à supprimer:

- tous les champs n'appartenant pas à un ensemble de champs donné

- tous les mots des champs restants n'appartenant pas à un ensemble de mots donné.

N.B. ce préfiltrage peut aussi être vu - et réalisé - comme une suite de deux filtrages: un sur les champs puis un sur les mots.

Alors, tout caractère peut être classé dans l'une des quatre catégories suivantes:

- délimiteur d'un champ à conserver
- délimiteur d'un champ à supprimer
- caractère de mot
- autre

De plus, on extrait du critère de filtrage l'ensemble des mots-clés qui y figurent, quelles que soient les conditions booléeennes s'appliçant sur leur présence. Par exemple, étant donnée la question:

'' trouver les articles contenant (PARIS et LILLE) ou (LYON et pas MARSEILLE) dans le champ Ville ''

on en déduit que le préfiltre ne va laisser passer que le champ Ville, et que les mots PARIS, LILLE, LYON et MARSEILLE dans ce champ.

Ces informations vont être indiquées dans deux tables:

-une table donnant le type de chaque caractère parmi les quatre types mentionnés plus haut:

* l'ensemble des caractères de mots est l'union des caractères des mots-clés; (cet ensemble est donc variable à chaque question; par exemple il contient le trait d'union si et

seulement si le trait d'union figure dans un mot-clè)
* pour chaque dèlimiteur de champ, on indique si le champ associè est à conserver ou non

Ces informations tiennent dans une table de N mots de 2 bits.

- L'ensemble des mots clès est reprèsentè ainsi: pour chaque mot-clè, on calcule un Hcode sur p bits, et l'on met à 1 dans une mèmoire de 2**p bits le bit dont l'adresse est ègale à ce Hcode, les autres bits ètant à zèro.

Le prèfiltre a l'architecture suivante:

- un filtrage des champs, qui se rèsume à ouvrir ou fermer un robinet: lorsque CE est un dèlimiteur de champ à conserver, alors on met (ou on laisse) le robinet en position ouverte. Lorsque CE est un dèlimiteur de champ à èliminer, on ferme le robinet. Les caractères sortant de ce premier filtrage entrent dans la station suivante dite de codage.

- la station de codage calcule le Hcode de chaque mot qu'elle reçoit, et indice la table des mots-clès avec ce code. Si un zèro est trouvè, alors le mot courant n'est pas envoyè en sortie.

La rèalisation matèrielle de cette station de codage peut être la suivante:

- le Hcode est calculè par un opèrateur travaillant en sèrie, caractère par caractère, qui dèlivre son rèsultat en permanence. La seule commande envoyèe à cet opèrateur est une remise à zèro avant le dèbut d'un nouveau mot.
- un mot commence lorsque le caractère prècèdent n'ètait pas de type ''mot'' et que le caractère courant l'est. Alors on commande la remise à zèro du calcul du Hcode.
- un mot se termine quand l'ancien caractère ètait de type ''mot'' et pas le nouveau. Alors l'indiciation de la table avec la valeur courante du Hcode indique si le mot qui se termine est à conserver ou non.

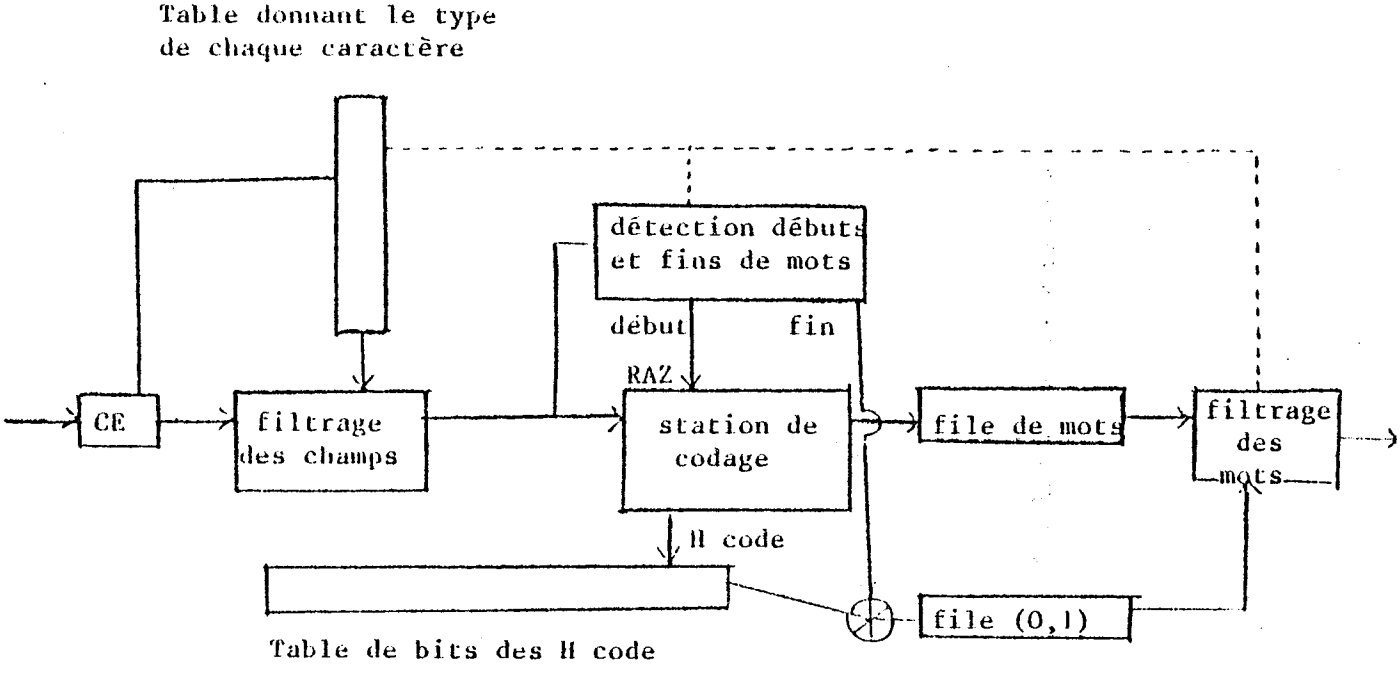
Comme on ne sait qu'à la fin du mot s'il est bon ou non, il

faut le mémoriser dans tous les cas. Pour cela on utilise deux boitiers FIFO: l'un dans lequel on envoie tous les mots, l'autre dans lequel on envoie le résultat 0 ou 1 attribué à ce mot. Ces deux files aboutissent à une station finale qui boucle sur l'algorithme suivant:

* lire la FIFO des résultats
* si on trouve un 0, lire le mot suivant dans la FIFO de mots
* si on trouve un 1, lire le mot suivant dans la FIFO de mots, et L'ECRIRE dans la FIFO de sortie finale du préfiltre.

En fin de compte, la sortie du préfiltre ne contiendra bien que les champs utiles au 'vrai' filtrage, et, à l'intérieur de ces champs, que les mots dont le Hcode est égal au Hcode d'un mot-clé utilisé par le critère de filtrage.

Voici un schéma général du préfiltre:



Cette architecture a bien les qualités nécessaires à un préfiltre:

- elle est très simple

- elle est surtout très rapide; une telle architecture peut accepter , avec des circuits du commerce, un débit d'entrée de plus de 10 millions de caractères par seconde, près de dix fois celui d'un disque standard. Un tel débit peut provenir par exemple de la sérialisation des débits de plusieurs têtes de disque fonctionnant en parallèle, et transportées par une fibre optique. Les disques optiques ont également un débit potentiel de cet ordre.

- elle réduit considérablement le débit de sortie. Avec des fichiers classiques, cette réduction dépend évidemment de la taille relative des champs conservés par rapport à celle de l'enregistrement. Mais avec des données textuelles, cette réduction peut probablement atteindre un facteur de 100 ou plus. Exemple: cela revient à ne retenir dans un article que les mots qui sont égaux à un codage près aux mots-clés de la question.

Une évaluation précise à priori du taux de réduction du débit apporté par le préfiltrage n'est pas facile à faire.

Ce taux dépend en effet de la fréquence des mots-clés recherchés dans le texte analysé, ainsi que du nombre de ces mots-clés.

Une indication sur ce dernier nombre est donnée dans [91], une étude statistique sur le système d'informations médicale américain MEDLINE. Il a été mesuré que le nombre moyen de mots-clés cités dans les questions des utilisateurs est de 3.

A ces mots-clés, le système documentaire rajoute des mots ayant un sens voisin, à l'aide de ses thésaurus.

Le nombre moyen total résultant est de 7 mots-clés.

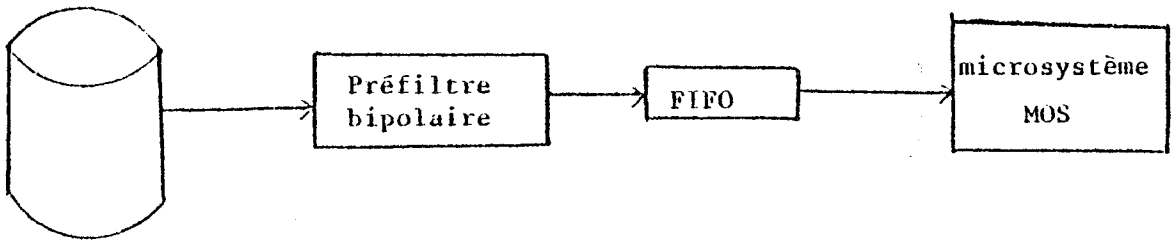
Ce nombre très faible laisse entrevoir un fort taux d'élimination lors du filtrage.

Quand à la fréquence d'occurence des mots-clés, elle dépend essentiellement de la taille de l'ensemble des mots considérés.

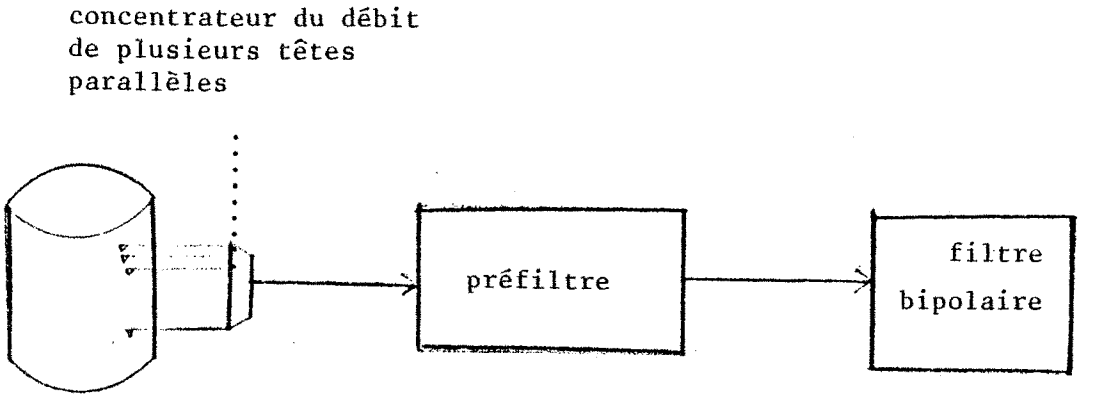
Si les données sont des textes d'une langue naturelle, ces fréquences seront plus faibles que si les données sont constituées d'un ensemble de mots-clés déjà présélectionnés par un documentaliste comme appartenant à une famille de mots bien définis, une liste de villes, de noms propres, de professions, etc...

La réduction du débit est exploitable de deux manières différentes:

- avec des débits ''moyens'' de 10 Mbits/sec, on peut remplacer un filtre construit en circuits rapides à faible intégration (comme la plupart de ceux proposés dans les machines bases de données actuelles) par un préfiltre de même technologie mais plus simple associé à un MICROPROCESSEUR MOS CLASSIQUE, qui va réaliser en LOGICIEL un filtrage par automates.



- avec des débits rapides de 100Mbits/sec, qui sont inexploitables par les filtres en technologie bipolaire des machines bases de données actuelles, il suffit de faire précéder ces filtres par un préfiltre:



3 6 QUELQUES VARIANTES DU PREFILTRAGE.

a) les Hcodes de chaque mot peuvent avoir été calculés à l'avance et stockés avec les données, en tête de chaque mot. Alors le préfiltrage des mots devient ultra-rudimentaire et se limite à laisser passer les mots dont le Hcode indice un 1 dans une table de bits.

b) un fichier textuel peut exister sous deux formes, l'une contenant le texte en clair, l'autre ne contenant que les Hcodes de chaque mot. Si l'on élimine les mots de moins de trois lettres, et si le Hcode est de 16 bits, (les blancs entre mots sont aussi supprimés), LA TAILLE MEMOIRE EST REDUITE QUATRE FOIS, (chiffre obtenu à partir de statistiques [28] sur la langue française). Ceci va correspondre à diviser par quatre le temps de balayage séquentiel du fichier, donc aussi le temps de réponse; C'est un très bon point en faveur des techniques de balayage par opposition à celles d'indexation et fichiers inverses.

Le résultat du balayage du fichier codé est de donner les références des enregistrements qui intéressent l'utilisateur, et qui seront accédés de manière classique sur la version en clair du fichier via ces références.

c) dans le cas d'applications ''grand-public'', on peut supprimer l'étape de filtrage sophistiqué et les critères booléens généraux. On peut par exemple se contenter de COMPTER le nombre de mots ayant un bon Hcode, le critère de recherche pouvant être de la forme: trouver les documents contenant au moins 5 mots parmi les mots indiqués. Le matériel devient très

simple et dès aujourd'hui concevable pour des applications comme la télédiffusion de textes ANTIOPE [34], où il est actuellement nécessaire de connaitre le numéro de l'enregistrement qui contient l'information désirée ... Un autre domaine d'application sera prochainement la consultation à domicile de données sur disques optiques à grande diffusion (catalogues par exemple)

d) problèmes de sous-chaines et d'orthographe.

Pour tolérer les fautes d'orthographe, il suffit de mettre à 1 non seulement le bit à l'adresse du Hcode, mais aussi ceux aux adresses égales aux Hcodes des mots déduits de l'original par une faute d'orthographe. Cela augmente évidemment le risque de collisions indésirables d'une manière qui reste à étudier.

Pour rechercher non seulement la coincidence du mot-clé avec un mot, mais son APPARTENANCE à un mot, il faudrait calculer le Hcode non seulement à la fin du mot, mais à chaque nouveau caractère du mot courant, et pour chaque sous-chaine du mot, puis faire le ou de tous les résultats obtenus par indiciation de la table de bits. Ceci demanderait beaucoup trop de calculs en général, et les algorithmes utilisant des automates sont plus simples et plus performants.

e) une suggestion de Hcode pour les données textuelles.

Une taille raisonnable du Hcode semble être 16 bits pour des raisons pratiques. La taille de la table est de 64K bits. Pour accélérer sa remise à zéro, elle peut être implémentée en parallèle, 4k mots de 16 bits par exemple. Pour coder un mot de la langue française sur 16 bits, il peut être intéressant de ne pas utiliser les techniques classiques de l'informatique dont le but est surtout d'éviter les collisions, mais de prendre en compte certaines caractéristiques linguistiques. Dans cet esprit, nous suggérons le codage suivant:

- on retient les 3 premières consonnes et les 2 premières voyelles de chaque mot

- chaque consonne est codée sur 4 bits en tenant compte de leur fréquence: les 15 premières sont codées de 0 à 14, et les 5 dernières (X,J,K,Z,W) sont toutes codées 15

- les voyelles sont codées ainsi:
  A → 0

E → 1
O et U → 2
I et Y → 3

- s'il y a moins de 3 consonnes ou de 2 voyelles, on complète avec le code le moins fréquent

Ce codage privilégie les consonnes et conserve leur ORDRE car ceci constitue le squelette de chaque mot. (L'alphabet phénicien n'avait que des consonnes, et cette propriété subsiste aujourd'hui dans les sigles des aéroports internationaux et dans les identificateurs des programmes des systèmes IBM ...)
Nous ne prétendons pas faire ici un travail de linguiste, mais suggérons plutot d'étudier de prés leurs travaux avant de spécifier un système de bases de données textuelles.

Ce codage est probablement améliorable; il a entrainé 4 collisions sur notre exemple des 320 prénoms:

- AYMAR et MARIE
- MARC et MAURICE
- GEORGES et GREGOIRE
- GASTON et AUGUSTIN

On peut aussi supprimer le doublement des consonnes, faire certains codages phonétiques ...
Il faut remarquer que le but de ce codage est ambigu, puisqu'il veut à la fois DISTINGUER des mots différents, et CONFONDRE des mots ressemblants.

3 7 UN FILTRAGE ENCORE PLUS ELEMENTAIRE: LE FILTRAGE DES CARACTERES OU LA FIFO INTELLIGENTE.

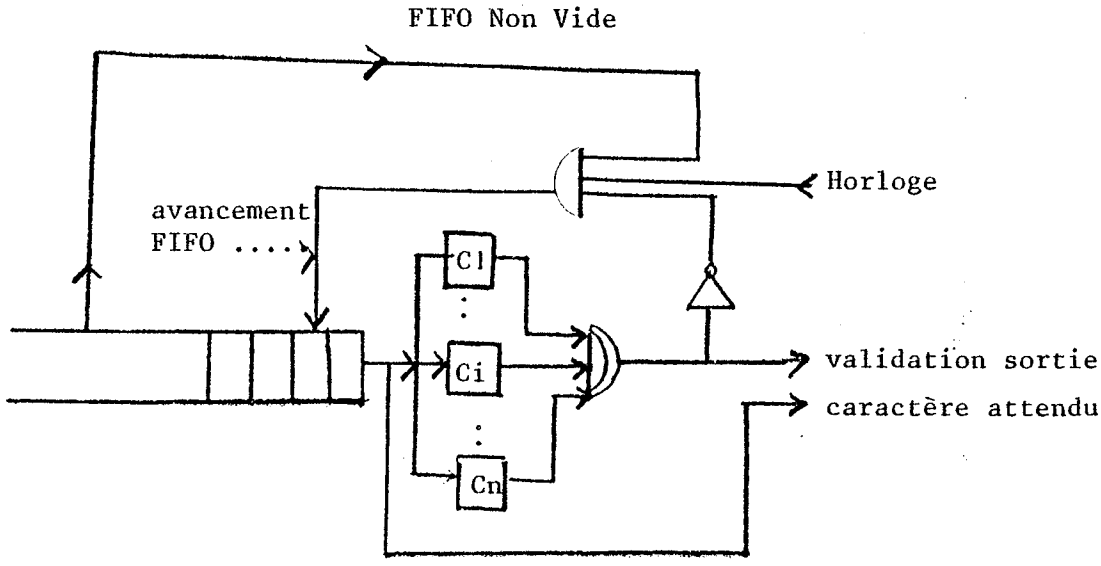
Pour accélérer la vitesse du filtrage, notre souci principal a jusqu'ici été de diminuer le nombre d'accès à la mémoire dans laquelle est représenté l'automate. Pour cela, nous avons préconisé des organisations dichotomiques ou tabulées des automates.

Par exemple, avec un automate tabulé, il suffit de faire un seul accès mémoire par caractère en entrée.

Il y a un moyen simple de faire mieux: une opération fréquente programmée par un automate est d'ATTENDRE un ou plusieurs caractères donnés, c'est à dire de lire le caractère suivant en entrée si le courant n'est pas égal à une des valeurs attendues. Avec nos automates actuels, il faut faire un ou plusieurs accès mémoire pour lire ces valeurs attendues.

Il est bien plus simple d'utiliser un mécanisme CABLE associé à la FIFO, auquel on envoie la valeur du caractère attendu. Ce mécanisme provoquera lui-même l'avancement de la FIFO, SANS TRANSMETTRE LES CARACTERES LUS AU FILTRE, tant que ces caractères sont différents du caractère attendu.

Si l'on attend plusieurs caractères, il suffit de posséder plusieurs couples (registre, comparateur):



On voit que les fonctions exécutées par la FIFO s'enrichissent:

- à la commande initiale 'LIRE LE CARACTERE SUIVANT' peuvent s'ajouter:

- RANGER UNE VALEUR DANS LE REGISTRE i

- ATTENDRE UN CARACTERE EGAL AU REGISTRES i OU j OU k, la liste sur laquelle porte le 'OU' pouvant par exemple être codée par un vecteur de bits transmis par la commande:

- CHARGER LE VECTEUR DE CHOIX DES REGISTRES AVEC LA VALEUR V

Ainsi, chaque fois que l'automate voudra attendre un ou plusieurs caractères, il lui suffira d'envoyer une ou plusieurs commandes à la ''FIFO INTELLIGENTE'' plutot que de faire un ou plusieurs accès mémoire à son programme.

Or cette situation est extrêmement fréquente lors des problèmes pratiques du filtrage:

- attente de la fin d'un champ qui n'est pas utilisé par la condition de filtrage

- attente de la fin d''un enregistrement dont on est déjà certain qu''il nous intéresse ou pas

- attente de la fin d'un mot dont on sait, au vu des premières lettres, qu'il n'appartient pas à l'ensemble recherché.

Nous avons fait quelques mesures sur ce dernier point, sur l'exemple des prénoms, (donc dans un univers très défavorable, puisque ne comportant que 320 mots différents).

- si l'on cherche la présence de 7 mots-clés (moyenne observée dans E 91 1), 79 pour cent des accès mémoire sont supprimés. Les mots faisant en moyenne 8 lettres, cela signifie que, en moyenne, il suffit de lire 1,6 caractères d'un mot pour l'éliminer et passer au suivant.
La réduction du débit obtenue par ce procédé rudimentaire est égale à 5. Le débit apparent d'un disque classique passe à un caractère toutes les 4 microsecondes, ce qui commence à le mettre à la portée de beaucoup de processeurs.

Un essai avec la recherche de 50 mots au lieu de 7 (ce qui est considérable) élimine encore 65 pour cent des accès mémoire, donc réduit le débit près de trois fois.

Il est important de noter que ce mécanisme travaille toujours sous le contrôle du filtre (ou du processeur qui en tient lieu), et qu'il ne diminue en rien la richesse des opérations permises par la technique des automates.

3 8 CONCLUSION SUR LES ASPECTS LEXICAUX DES AUTOMATES

Nous avons rappelé que la recherche de mots-clés dans un texte est réalisable grâce à la construction de l'automate de reconnaissance de ces mots-clés, et que la réalisation matérielle d'opérateur capable d'exécuter cet automate est très simple. Cette solution est meilleure que toutes les autres techniques proposées dans les architectures de bases de données connues:

- elle demande moins de logique de traitement, puisque séquentielle par nature. Un seul comparateur suffit contrairement aux machines TGI [20], DBC [21], DIALOG [25]

- la recherche est effectuée en une seule passe, indépendamment du nombre de mots-clés recherchés, contrairement aux machines CASSMC [24], RAP [25]

- les mémoires utilisées pour contenir l'automate peuvent être à la fois lentes et étroites.

- les comparaisons < et > sont possibles, contrairement à la machine AFP [26], qui utilise aussi les automates. Ce point est très important car il permet de traiter non seulement les données textuelles mais aussi les fichiers habituels de l'informatique de gestion

- différentes techniques de représentation et d'exécution des automates ont été discutées. En particulier:

- le temps de compilation de l'automate peut être LINEAIRE, et même s'effectuer AU VOL, alors qu'il est très lent dans AFP [26].

- en conséquence du point précédent, nous avons proposé la première architecture -à notre connaissance- pour effectuer AU VOL la jonction sur égalité, résultat qui sera généralisé plus loin pour le tri, l'union, l'intersection et la projection.

- différentes techniques de représentation et d'exécution

des automates ont été discutées. Elles permettent d'envisager des architectures très variées, correspondant à différents compromis: prix, performance, technologie MOS ou bipolaire, applications professionnelles ou grand-public.

Compte tenu de l'état actuel et prévisible de la technologie, la solution du filtrage par automates semble nettement préférable aux autres solutions.

Cette conclusion du chapitre consacré uniquement aux problèmes lexicaux va être confirmée et amplifiée par les trois chapitres qui suivent et traitent des:

- aspects sémantiques: calculs booléens, arithmétiques ou statistiques

- aspects syntaxiques: reconnaissance des structures internes des enregistrements, processus coopérants de filtrage

- principaux exemples d'applications des filtres: fichiers séquentiels ou indexés, bases de données.

CHAPITRE 4

TECHNIQUES POUR RESOUDRE LES PROBLEMES SEMANTIQUES DU FILTRAGE

TECHNIQUES POUR RESOUDRE LES PROBLEMES SEMANTIQUES DU FILTRAGE

4 1 INTRODUCTION: DU FILTRAGE AU CALCUL

Le but initial d'un filtre est de filtrer, c'est à dire de laisser passer ou non un enregistrement. Cela revient à considérer le filtrage comme l'application à chaque enregistrement E d'une fonction booléenne F. L'enregistrement est conservé si et seulement si F(E) vaut 1.

Ceci correspond à la notion habituelle d'opération de sélection sur une base de données: '' trouver toutes les personnes dont l'age est > 50 et le salaire < 1000''

On peut généraliser ce processus en considérant que le résultat du filtrage peut être quelconque et pas seulement booléen. Par exemple ce résultat peut être un nombre entier ou réel que nous appelons la NOTE attribuée par le filtre à chaque enregistrement. Si l'on considère un ensemble d'enregistrements (fichier), le filtre va fournir l'ensemble des notes de ces enregistrements; l'utilisateur peut alors poser des questions à

la sémantique beaucoup plus riche qu'un simple filtrage booléen, par exemple:

- maximum, minimum, moyenne des notes

- histogramme des notes

- donner les n enregistrements ayant les meilleures notes.

Le point fondamental est que l'on commence à quitter le strict point de vue de l'informatique ''informaticienne'' (accéder à un fichier, faire une machine qui respecte un langage d'interrogation existant et normalisé), pour tenter d'aborder le problème de l'utilisateur final: classifier, synthétiser, analyser des données soit pour prendre des décisions, soit pour savoir quelles données communiquer à qui ...

Plus concrètement, l'intérêt de la notion de NOTE est clair en informatique documentaire. Si l'on cherche les articles contenant le mot PARIS et qu'il y en a dix mille, on préfèrera s'intéresser à ceux qui le contiennent le plus grand nombre de fois. Le critère de filtrage sera: compter le nombre d'occurences du mot PARIS dans chaque article, et afficher les articles ayant les 20 meilleures notes, la note étant ce nombre d'occurences.
Le problème à résoudre dans ce cas est que la décision de garder ou non un enregistrement dépend non seulement de sa propre note, mais aussi des autres notes. Ceci pose de gros problèmes de gestion mémoire auquels des solutions ont été apportées dans la machine TREFLE par la création d'un module spécifique, [37], [38],[39]

Une manière encore plus générale de considérer le filtrage est de dire que le résultat de F(E) est lui-même un enregistrement quelconque.
Ceci recouvre l'opération de projection de l'algèbre relationnelle, qui revient à supprimer certains champs de E, et qui peut être généralisée à la projection CONDITIONNELLE, comme dans l'exemple suivant:

F(E) → SI champ AGE < 18, alors champ ADRESSE DES PARENTS

F(E) → SI champ AGE ≥ 18, alors champ ADRESSE PERSONNELLE

Un autre aspect sémantique important pour le traitement des données textuelles est la tolérance aux fautes d'orthographe. Un chapitre spécial lui est consacré plus loin.

La grande question que ce chapitre voudrait poser - et tenter de résoudre partiellement- est une question fondamentale:

* les machines bases de données (dont on dit qu'elles ''arrivent'' [27]) ont-elles un avenir?

Si l'on admet qu'il leur suffit de trouver des acheteurs pour avoir un avenir, on peut poser deux sous-questions:

* les machines base de données seront-elles capables d'accélérer la vitesse d'exécution des applications EXISTANTES, écrites pour la plupart en COBOL, et utilisant des langages d'interrogation 'navigationnels' ou même pas de bases de données du tout, mais seulement des primitives d'accès à des fichiers?
Ce nouveau problème présente une étroite analogie avec une vieille question: '' peut-on accélérer la vitesse d'exécution de programmes FORTRAN EXISTANTS avec un multiprocesseur? ''
Cette analogie n'a pas de quoi nous rendre optimiste ... Le problème est néanmois différent. Là, il s'agit de PARALLELISER, ici il s'agit de SEQUENTIALISER. Nous l'examinerons un peu dans la conclusion.

* la deuxième question est: existera-t-il un jour un langage qui supprimera la dichotomie actuelle entre:

- langage de PROGRAMMATION (COBOL, PASCAL)

- langages d'accès aux fichiers (Read, Write, algèbre relationnelle)

Ce qui est nécessaire, c'est un LANGAGE DE TRAITEMENT DES DONNEES qui ne coupe pas le problème en petits morceaux (masqués par le nom d'interfaces) et qui oblige à utiliser un langage pour accéder aux données et un autre pour faire les calculs.

Ce qui pourrait sembler ne pas être un mal en soi (on divise les problèmes, on diminue les risques d'erreurs) a en fait un grand nombre de mauvaises conséquences:

- la division des langages entraine une division du travail des concepteurs et réalisateurs. Personne n'a une vue d'ensemble du ''monstre logiciel'', même l'utilisateur l'ignore, sauf en ce qui concerne le temps de réponse ...

- il semble quasi impossible d'imaginer un compilateur capable , à partir d'une application écrite en COBOL et DL/1 ou COBOL et SOCRATE, de générer un code divisé en deux parties, l'une s'exécutant sur un ordinateur hote classique, l'autre sur un processeur base de données effectuant le maximum de traitements au vol. Il faudrait que ce compilateur ''casse'' une interface logicielle (entre COBOL et le langage d'accès) et en crée une autre, matérielle (entre l'ordinateur général et la machime base de données).

Tant qu'un tel langage n'existera pas, il ne faudra pas attendre de grandes évolutions dans l'architecture des ordinateurs UNIVERSELS. Les améliorations comme celles présentées dans cette thèse resteront difficiles à mettre en oeuvre hors de cas particuliers.

Les deux langages s'éloignant le moins de cet idéal sont aujourd'hui APL et à un moindre titre SOCRATE. Mais l'un et l'autre ne sont pas assez ASSOCIATIFS. L'algèbre relationnelle est un bon langage associatif, mais pas assez général en ce qui concerne les calculs autorisés.

Ces considérations , pour philosophiques qu'elles soient, limitent très concrétement la portée des travaux actuels en architecture de machines bases de données. C'est en ayant présente à l'esprit cette limitation qu'il faut lire les paragraphes qui suivent, et qui sont consacrés aux évolutions des solutions successives proposées par l'auteur au problème des évaluations sémantiques sur des données balayées au vol.

Après l'analyse qui précède, le lecteur ne sera pas étonné de constater que l'évolution des architectures proposées va dans le sens d'une plus grande banalisation - voire banalité ... - des unités de calcul, l'originalité se retrouvant au niveau de la gestion des flots de données circulant entre les différentes

unités.

Nous exposons successivement trois solutions:

- le calcul intègrè au filtre
- un module de calcul indépendant du filtre
- le filtre intègrè au module de calcul

4 2 FONCTIONS DE CALCUL INTEGREES AU FILTRE

En poursuivant notre analogie avec la compilation, on peut considérer les traitements et calculs à effectuer lors du défilement des données dans le filtre comme l'équivalent des actions sémantiques: lorsque l'analyse du texte source a atteint tel ou tel état, on exécute certaines opérations. Dans notre cas, les états intéressants atteints par le filtrage sont par exemple la reconnaissance d'un mot. Cette reconnaissance a pu se faire avec un automate compilé dans un langage dont l'instruction de base est de la forme:

VC	A=	A<	A>

Cette instruction n'a pour but que de calculer l'adresse de l'instruction suivante: A= ou A< ou A> suivant que CE est = ou < ou > à VC.

Une solution naturelle est de compléter cette instruction qui ne fait que du séquencement par une seconde partie qui fait du calcul:

VC	A=	A<	A>	codage du calcul

On aboutit à une structure habituelle de microinstruction horizontale. Le calcul spécifié dans ce nouveau champ n'étant exécutée que si CE = VC.

Ainsi, parallélement à la fonction de séquencement, on peut exécuter une action sémantique, qui durera le même temps et ne ralentira pas le filtrage. Quelles actions peut-on exécuter en un aussi court laps de temps?

4 2 1 LE CALCUL BOOLEEN

Le calcul booléen est l'action la plus répandue dans les langages d'interrogation, c'est souvent même la seule forme de ''calcul'' qu'ils permettent: on exprime le critère de filtrage comme une expression booléenne portant sur la présence ou

l'absence de mots-clès dans l'enregistrement analysè:

''selectionner les articles contenant (PARIS ou MARSEILLE) et (LILLE ou LYON) et pas NICE ''

Chaque fois que la partie sèquencement du filtre dècouvre l'un de ces mots, on dispose d'une information utile à l'èvaluation de l'expression boolèenne. Une autre information utile est la dètection de la fin de l'enregistrement; alors on peut connaitre le rèsultat final. Notre principale contrainte est de faire le filtrage AU VOL. On a vu des solutions pour reconnaitre les mots-clès au vol, indèpendamment du nombre de ces mots-clès. Si l'on utilise les techniques habituelles de calcul pour èvaluer les expressions boolèennes (c'est à dire leur compilation ou interprètation sous forme postfixèe) deux problèmes se posent:

- le temps de calcul n'est pas indèpendant de la complexitè de l'expression: il est proportionnel au nombre d'opèrateurs boolèens de l'expression.

- l'ordre des calculs dèpend de la structure de l'expression et de l'ordre des opèrations dans le code postfixè, alors que l'ordre d'occurence des mots-clès dans l'enregistrement (donc des opèrandes de l'expression) est imprèvisible et variable d'un enregistrement à l'autre.

Dans le cas gènèral, il faudra attendre la fin de l'enregistrement pour COMMENCER le calcul de l'expression postfixèe, et ce calcul aura une longueur qui dèpend de la complexitè de l'expression. Il y a donc en gènèral impossibilitè d'effectuer ce calcul au vol; il risque de n'être pas terminè quand l'enregistrement suivant arrivera.

Dans ces conditions,une alternative très simple consiste à reporter l'essentiel du travail au moment de la COMPILATION du critère de filtrage. La fonction boolèenne va être calculèe une fois pour toutes avec toutes les configurations possibles de ses opèrandes et TABULEE.

Si la fonction boolèenne a p variables, elle peut être reprèsentèe par sa table de vèritè sous forme d'une mèmoire de 2**p bits. Pour chacune des 2**p combinaisons des valeurs de ces

variables, on range la valeur correspondante de la fonction dans le bit dont l'adresse est la concaténation des p variables.

On attribue à chaque mot-clé un numéro de variable de 0 à p-1. La variable i vaut 1 si et seulement si le mot-clé i est présent. Exemple, pour p = 4:

(PARIS OU LILLE) ET NON (NICE ET MARSEILLE)

PARIS →→ b0
LILLE →→ b1
NICE →→ b2
MARSEILLE →→ b3

Table de vérité:

b3	b2	b1	b0			b3	b2	b1	b0	
0	0	0	0	→	0	1	0	0	0	→ 0
0	0	0	1	→	1	1	0	0	1	→ 1
0	0	1	0	→	1	1	0	1	0	→ 1
0	0	1	1	→	1	1	0	1	1	→ 1
0	1	0	0	→	0	1	1	0	0	→ 0
0	1	0	1	→	1	1	1	0	1	→ 0
0	1	1	0	→	1	1	1	1	0	→ 0
0	1	1	1	→	1	1	1	1	1	→ 0

Ce travail est fait à la compilation du critère de filtrage, et rangé dans une mémoire vive.

A l'exécution, les choses se passent ainsi:

- il existe un vecteur de p bascules individuellement adressables

- ces bascules sont mises à zéro au début de chaque enregistrement

- quand le mot i est reconnu, la bascule numéro i est mise à 1

- à la fin de l'enregistrement, l'indiciation de la table de vérité par le vecteur des bascules donne le résultat de

l'expression booléenne.

Ce mécanisme très simple respecte bien notre contrainte principale: l'évaluation de l'expression prend un temps indépendant de sa complexité. La contrainte devient spatiale: il faut une table de vérité de 2**p bits.

Remarquons que cette technique ne limite pas à p le nombre de mots-clés. En effet, tous les mots-clés se trouvant dans un ''ou'' comme ''LUNDI ou MARDI ou JEUDI ou SAMEDI'' vont avoir évidemment le même numéro; (cette situation est très fréquente en informatique documentaire: listes de synonymes). Pour les ni (ni LUNDI ni MERCREDI ni DIMANCHE), le même numéro peut aussi être attribué aux mots-clés.

Pour les ''et'', et afin de ne pas avoir à attribuer une bascule à chaque mot-clé, il faut compliquer le matériel, et associer un COMPTEUR à chaque monome. Au début, on remet le compteur à zéro et on l'incrémente chaque fois qu'un mot est trouvé. On teste alors si sa valeur est égale au nombre total de mots-clés, et si oui, on positionne La bascule associée à ce monome à 1. Ceci ne marche que si on est sûr que chaque mot ne sera trouvé qu'une fois, ce qui limite l'application de ce mécanisme aux fichers qui ne contiennent pas du texte libre, mais par exemple des listes explicites de mots-clés.
Plus généralement, ce mécanisme permet d'introduire des conditions de la forme AU PLUS, AU MOINS.

Une autre question à examiner est le temps pris par la compilation de la table de vérité. Une solution naive est de générer les 2**p combinaisons et d'évaluer 2**p fois la fonction. Si p est grand, ce temps de calcul, s'ajoutant à celui de compilation de l'automate, peut devenir critique. Il existe une autre solution simple dont la complexité est plus faible:

- étant donnée une expression, on numérote chacun des mots-clés

- on remplace chaque mot-clé de numéro i par un vecteur de bits Vi tel que Vi[k] = i eme bit de la représentation binaire de

k. Exemple:

* V0 = 0 1 0 1 0 1 0 1 0 ...
* V1 = 0 0 1 1 0 0 1 1 0 0 1 1 ...
* V2 = 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 ....

- la table de vérité est le résultat de l'expression booléenne effectuée bit à bit sur les Vi. Comme la plupart des machines disposent d'opérations booléennes bit à bit portant sur un mot, ce calcul est très rapide d'autant que les valeurs des Vi sont des constantes précalculées.

Exemple:

(PARIS ou LILLE) et non (NICE et MARSEILLE)

se transforme en:

(V0 ou V1) et non (V2 et V3)

```
          0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
  ou
          0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
et
          0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
  non et
          0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

→→→       0 1 1 1 0 1 1 1 0 1 1 1 0 0 0 0
```

Sur un processeur à mots de 16 bits à une adresse et à accumulateur, le code est simplement:

```
LOAD V0
OR V1
STORE M
LOAD V2
AND V3
NOT
AND M
STORE RESULTAT
```

Au niveau de l'instruction du filtre, il faut donc prévoir les actions suivantes:

- remettre à zèro le vecteur de bascules

- positionner la bascule numèro i

- obtenir le rèsultat de l'indiciation de la table de vèritè par les bascules

Si certaines bascules sont associèes à des compteurs (pour rèsoudre efficacement les ''et''), il faut pouvoir dans ce cas:

- remettre à zèro la bascule et le compteur

- incrèmenter le compteur i, le comparer à une constante et positionner la bascule i en fonction de cette comparaison

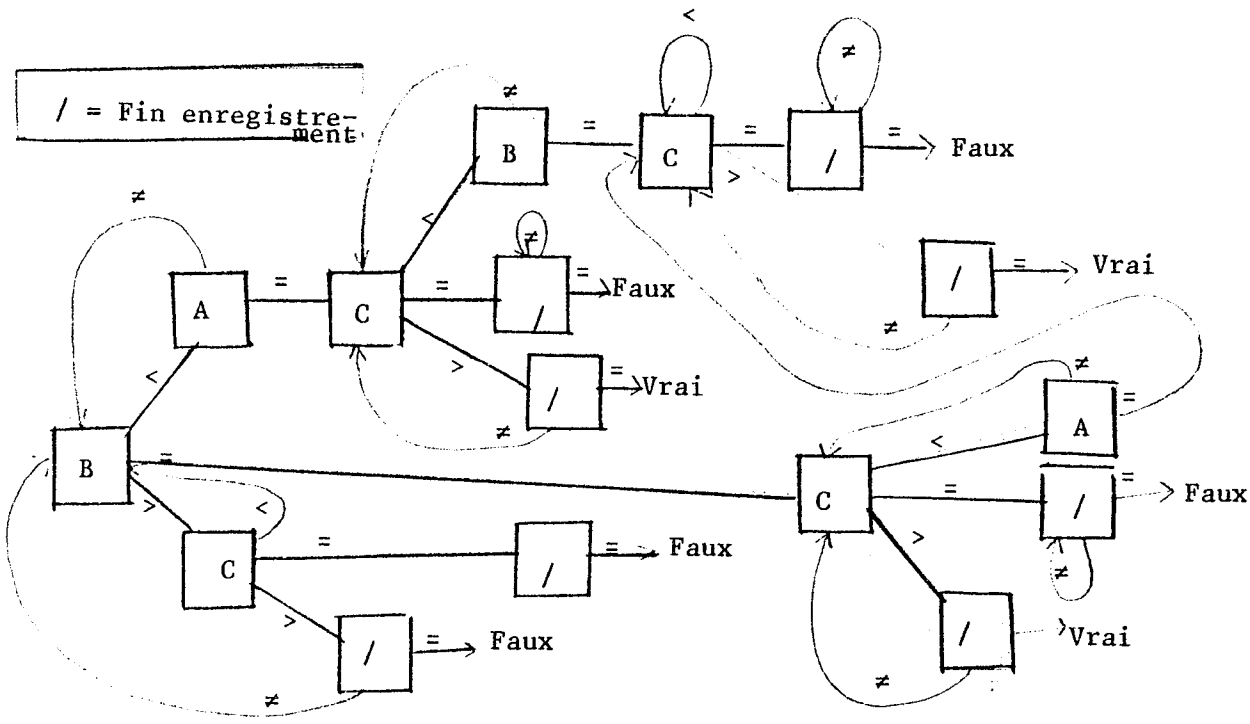
L'èvaluation des conditions boolèennes est donc très simple et très rapide avec la solution des tables de vèritè prècompilèes. Cette technique n'avait jamais èté encore proposèe dans les machines base de donnèes qui doivent en gènèral soit effectuer autant de passes qu'il y a d'opèrations boolèennes (cf CASSM, RAP), soit dècomposer de manière couteuse la question sous forme normale disjonctive et avoir un module de comparaison par monome (cf DBC)

Il faut aussi rappeler que la technique de filtrage par automates permettait dèjà à elle seule de rèsoudre le OU sur la prèsence de mots-clès.

Un auteur [42] nous a indiquè que , plus gènèralement, n'importe quelle fonction boolèenne portant sur la prèsence ou l'absence d'un mot dans un texte peut être èvaluèe par un automate.

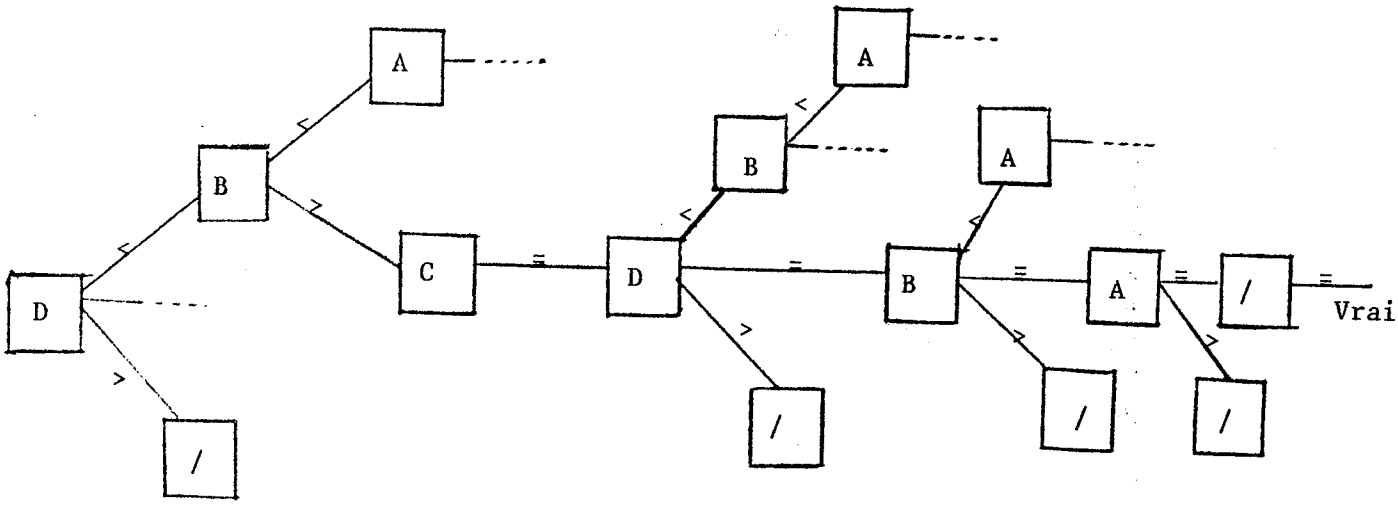
Exemples:

(A et B et non C)



(A et B et C et D)

Extrait de l'automate



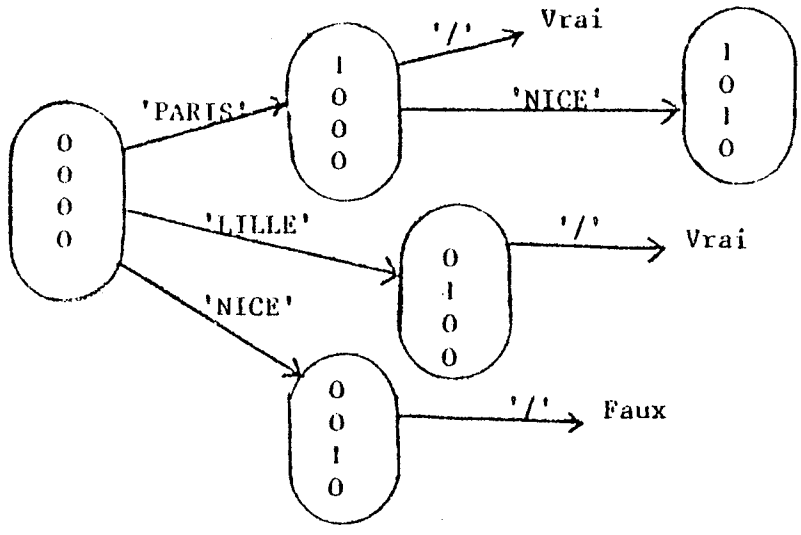
Suivant une suggestion de [90], une manière systématique de construire l'automate est d'associer un état à chacune des 2**n valeurs du vecteur des n conditions booléennes élémentaires, et d'exprimer que la rencontre d'une lettre fait passer à l'état déduit de l'état courant en positionnant à 1 le bit associé à la

lettre rencontrée.

Lorsque le délimiteur de fin est rencontré, le résultat de l'évaluation n'est autre que celui donné en adressant la table de vérité de la condition booléenne par la valeur associée à l'état courant.

Voici par exemple le début de l'automate correspondant à l'exemple précédemment traité par l'utilisation de bascules indiquant une table de vérité:

(PARIS ou LILLE) et non (NICE et MARSEILLE)



Ceci est un résultat théorique intéressant, mais en pratique il conduit à une explosion de la taille de l'automate et de son temps de compilation. De plus, en poursuivant dans la même voie, on peut aussi implémenter des compteurs, des additionneurs sous forme d'automates d'états finis ... L'utilisation des boitiers du commerce semble tout de même préférable!

4 2 2 CALCULS NON BOOLEENS

En informatique documentaire, le calcul booléen n'est pas très satisfaisant. Le calcul booléen est une notion d'informaticien, pas de documentaliste. La réponse qui intéresse l'utilisateur n'est pas ''tout noir ou tout blanc''. Il faut être plus nuancé et mesurer l'intérêt d'un article avec plus de finesse que 0 ou 1.

Une extension naturelle est de passer de la logique binaire à la logique k-aire. Le problème est que si l'on applique la technique des tables de vérité précédente, leur taille devient égale à (log2 k)×(k**p) bits, ce qui n'est pas envisageable. Par exemple, pour k = 8 et p = 8, la table contient 48 millions de bits.

En fait, deux types de problèmes se posent:

- peut-on faire des calculs non booléens au vol ?

- quels sont les calculs qui intéressent l'utilisateur ?

Pour répondre à la première question, il est clair que beaucoup de calculs ne peuvent s'effectuer au vol, c'est à dire en un temps proportionnel à la longueur des données: multiplications, divisions, fonctions transcendantes. On sait seulement faire en série les additions, soustractions, opérations booléennes bit à bit, les comparaisons.

Quand à l'utilisateur, tous les calculs l'intéressent, mais il ne veut pas savoir lesquels sont calculables au vol et lesquels ne le sont pas. Il jugera au temps de réponse, par comparaison aux autres machines, et non par comparaison au débit de son disque.

De plus, dans le cas de données textuelles, l'utilisateur n'a pas toujours une notion très claire de la manière dont un document l'intéresse. Comment tenir compte de la fréquence des mots, de la quantité d'information que chacun apporte. Si la question mentionne certaines conditions sur le Titre, d'autres sur l'Auteur, d'autres sur la Date, si certains mots-clés ont plus d'importance que d'autres pour le questionneur, par quel

calcul peut-on synthétiser tous ces points sous la forme d'une NOTE finale?

Une chose semble claire: il faut a priori pouvoir faire n'importe quel calcul, même lentement, plutot que de se limiter arbitrairement à ceux qui se font au vol.

Dans ces conditions, la solution précédente où les calculs sont intimement intégrés au parcours de l'automate au sein de chaque instruction n'est pas bonne pour les raisons suivantes:

a) raison quantitative: il est inutile de faire une action sémantique à chaque action de séquencement.

Ceci apparait clairement pour le calcul booléen. Dans chaque instruction on prévoit entre autres la possibilité de positionner une bascule et d'indexer la table de vérité, or la première action n'a lieu que lorsqu'un mot clé est trouvé (peut-être toutes les millisecondes ...) et la seconde à la fin de chaque enregistrement seulement. Il y a un gaspillage en circuits de calcul, mais surtout en circuits de mémoire, puisque plusieurs dizaines de bits sont lus à chaque cycle machine totalement inutilement. La mémoire correspondante pourrait être plus lente, et ses mots plus étroits et moins nombreux.

b) raison qualitative: il faut tout mettre ou ne rien mettre dans le filtrage.

Supposons que l'on veuille ajouter la faculté triviale de COMPTER ou CUMULER à un filtre. La solution ''électronique'' est de rajouter quelques boitiers de comptage, un ou deux registres et additionneurs, et les bits de commande correspondants dans l'instruction. Mais sur combien de bits faut-il compter, faut-il cumuler en binaire, en BCD, en flottant ...? Quelle que soit la solution câblée, il y a fort à parier qu'une utilisation imprévue surgira vite, qui, bien que très simple, sera impossible à satisfaire.

De telles considérations nous ont amené à DISSOCIER complètement la fonction de reconnaissance du filtre de sa fonction de calcul. Chaque fois que le filtre aura atteint un point intéressant (reconnaissance d'un mot-clé par exemple) , il se contentera de porter ce fait à la connaissance d'un autre

module, appelé MODULE DE CALCUL, qui est une machine UNIVERSELLE et PROGRAMMABLE, et se chargera d'effectuer les calculs associés à la reconnaissance de ce mot-clé.

4 3 UN MODULE DE CALCUL INDEPENDANT DU FILTRE

Ce chapitre décrit l'architecture d'un module de calcul et explique comment on peut le généraliser.

4 3 1 PRINCIPES D'UN MODULE DE CALCUL ASSOCIE A UN FILTRE

Le module de calcul (MC) doit être une machine universelle, rapide et programmable. l'architecture décrite ici a effectivement été réalisée. [43].

L'architecture choisie est une micromachine opérant sur des octets, dont l'instruction est longue (44 bits) et possédant une mémoire de données de mots de 8 bits. Une unité arithmétique et logique reçoit deux opérandes:

- le premier provient soit de la mémoire de données (MD) soit d'un registre DATA chargeable de l'extérieur

- le second provient de MD adressée soit directement par l'instruction, soit indirectement par un registre RX, qui est incrémentable.

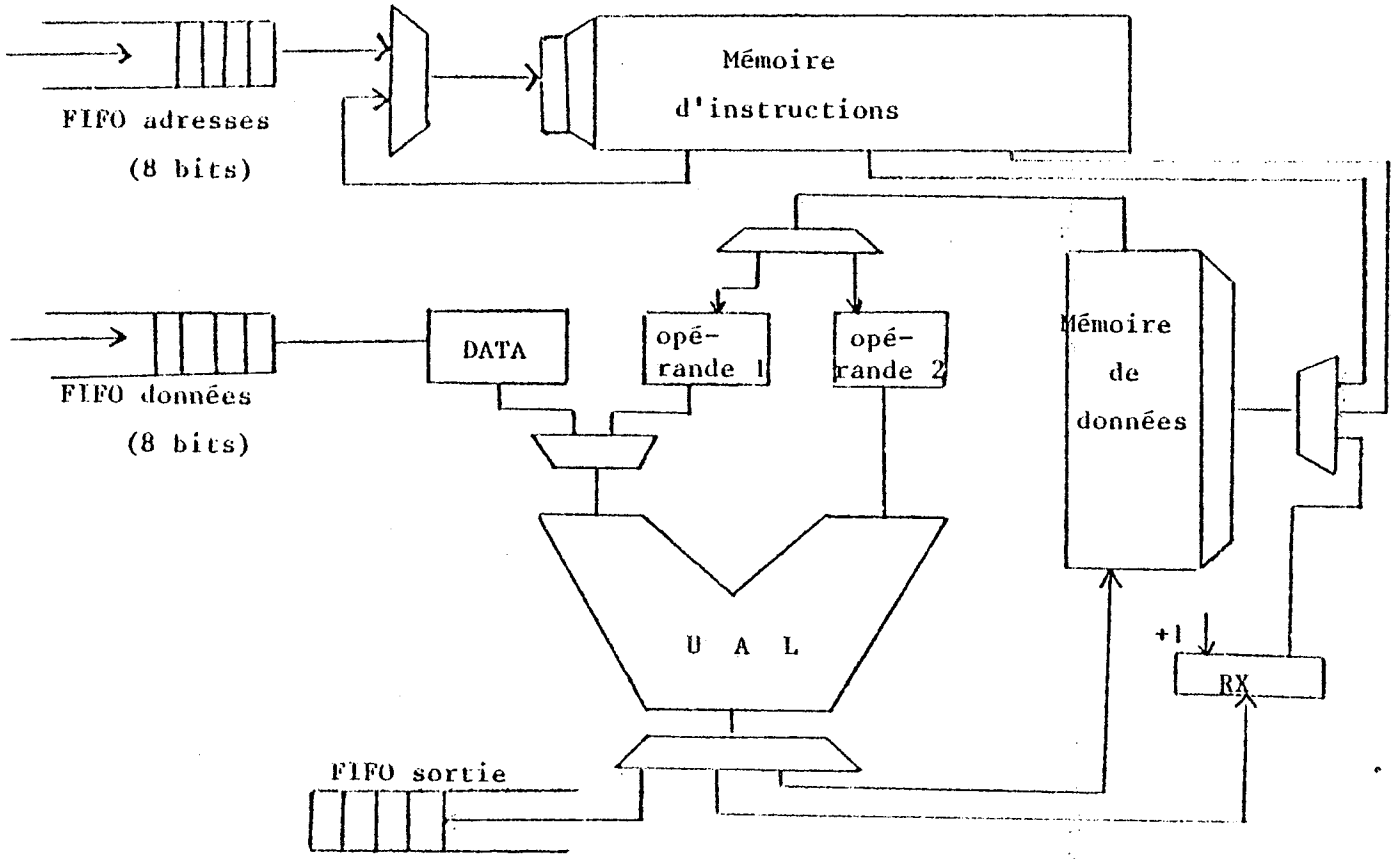
- le résultat de l'UAL est toujours rangé à l'adresse du deuxième opérande, et optionnellement dans RX et dans une FIFO de sortie.

- le calcul de l'adresse de l'instruction suivante est fait soit de la manière traditionnelle (branchements conditionnels ou inconditionnels) soit d'une manière spécifique au MC: l'adresse suivante peut provenir d'une FIFO remplie par le filtre.

Plus précisément, quand le mode d'adressage précise ''lire l'adresse suivante depuis la FIFO'', alors:

* l'horloge machine est bloquée jusqu'à ce que la FIFO d'entrée soit non vide.

* cette FIFO a 16 bits de large. 8 de ces bits donnent la nouvelle valeur du compteur ordinal du MC, et les huit autres sont chargés dans le registre DATA, qui peut être utilisé comme opérande des instructions du MC.



Chaque fois que le filtre veut donner du travail au MC, il lui envoie donc via une FIFO de 16 bits de large:

- une adresse de programme dans sa mémoire d'instructions

- une donnée

Cette donnée peut être soit:

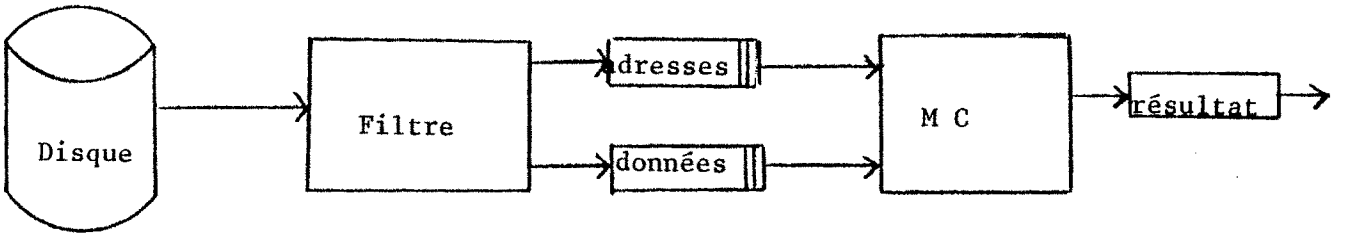
* une valeur immédiate contenue dans l'instruction du filtre

* la valeur de CE, le dernier caractère lu en entrée du filtre

Ce dernier point comble une lacune du filtre, qui ne peut actuellement ressortir les caractères qu'il lit, et va permettre au MC de faire des projections, des comptages, des histogrammes, des comparaisons entre champs d'un même enregistrement.

Le fait de mettre une file d'attente entre le filtre et le MC permet à ce dernier de rattraper le temps perdu quand il le peut, et de travailler de manière totalement asychrone par rapport au filtre.

L'ensemble filtre + MC peut être vu comme une architecture de machine assez originale, où le filtre joue le role d'un séquenceur associatif observant un flot de données, à partir duquel il crée un flot de commande qu'il envoie vers le MC.



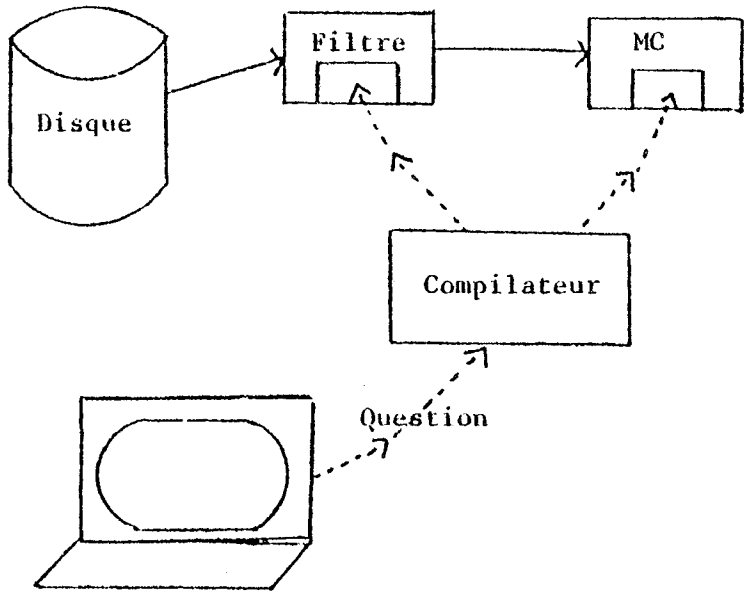
4 3 2 EXEMPLES D'UTILISATION DU MODULE DE CALCUL

A partir d'une requête de l'utilisateur, il faut maintenant compiler DEUX PROGRAMMES:

- l'automate de reconnaissance pour le filtre

- le programme de calcul pour le MC

- et mettre dans le premier les ordres d'envoi des couples (adresse, donnée) initialisant le deuxième::



4 3 3 Calculs booléens.

On garde notre solution tabulée, mais elle est réalisée de manière logicielle par le MC.

Quand il reconnait un mot-clé, le filtre envoie vers le MC le codage 1 parmi 8 de son numéro, et l'adresse d'un programme qui fait:

VBASC ← VBASC ou DATA (ce qui simule le positionnement d'une bascule)

Quand il reconnait la fin de l'enregistrement, le filtre envoie l'adresse d'un programme qui indice la table de vérité

rangée dans sa mémoire de données avec VBASC, obtenant ainsi le résultat booléen. Tout ceci va être un peu plus lent que dans un filtre, mais largement assez rapide pour la fréquence à laquelle ces événements se produisent. Et l'on dispose de toute la souplesse de la programmation pour optimiser tel ou tel point, sans investir dans du matériel, comme l'adjonction de compteurs pour résoudre les ''et'' entre de nombreux mots-clés.

4 3 4 Comptages, cumuls, histogrammes

Pour chaque article, on calcule une note obtenue en comptant 10 chaque fois que l'on trouve PISE, 5 quand on trouve ROME, et 4 NAPLES.

Chaque fois que le filtre trouve un de ces mots, il envoie le chiffre associé à un unique programme qui fait:

Note ← Note + DATA

Compter le nombre d'occurences de ces mêmes mots serait aussi facile.

Pour cumuler les valeurs d'un champ donné dans tout un fichier, il suffit que le filtre:

1) détecte le début de ce champ et envoie l'adresse d'un programme qui fera: RX← adresse du total

2) envoie au MC la valeur de ce champ octet par octet, poids faibles d'abord, avec l'adresse d'un programme qui fera:

- MEM[RX]←MEM[RX] + DATA + CARRY;
- RX←RX + 1

Pour faire des histogrammes, il suffit d'utiliser la valeur reçue dans DATA pour calculer l'adresse du compteur à incrémenter.

4 3 5 Calcul de notes non binaires en recherche documentaire.

Soit un critère de la forme:

((Age > 50) et (Auteur contient MAX ou LOUIS ou JULES)) ou non (Titre contient VILLE et CAMPAGNE)

Supposons que, par un moyen quelconque, la note 7 sur 10 a déjà été attribuée au champ Age, 5 à auteur et 6 au Titre. Comment calculer une note unique finale, sans poser de problème à l'utilisateur ?. Une solution simple est de remplacer les opérations binaires par leur correspondant en algèbre 10-aire:

Et →→ minimun
OU →→ maximum
Non →→ complément à 10

Alors notre note est (7 min 5) max (10 - 6), soit 5.

Cette technique est simple, et semble intuitivement respecter la sémantique supposée de la question.
Le MC possède ces opérations MAX et MIN dans son jeu d'instructions.

4 4 VERS DES TRAITEMENTS GENERAUX

Nous introduisons d'autres exemples de traitements désirables et examinons leur conséquence sur l'architecture de la machine.

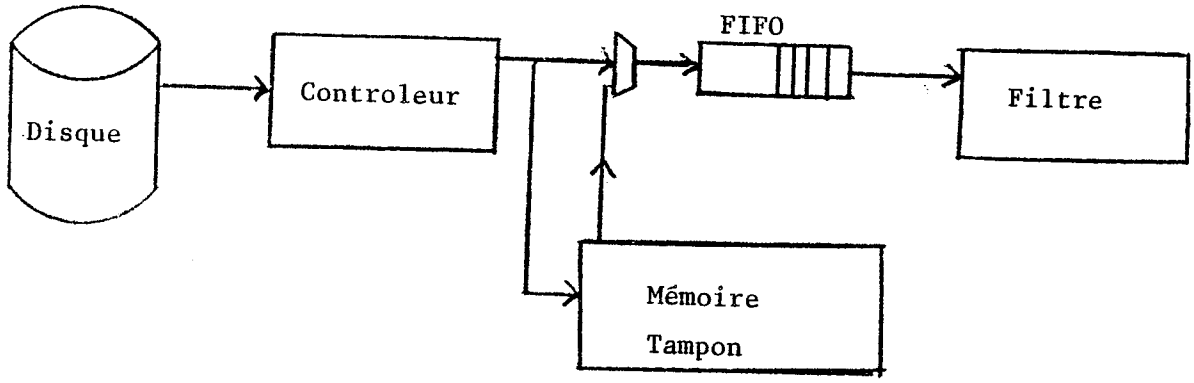
Soit un fichier dont les enregistrements contiennent entre autres les champs (Nom,Age,Salaire,Ville)
On recherche le nom des personnes dont le salaire est supérieur à K1 + (AGE x K2) et qui habitent à MARSEILLE. Il va falloir faire une multiplication dans le module de calcul. Aura-t-on le temps de faire ces multiplications au vol ? Tout va dépendre de leur fréquence, donc de la longueur des enregistrements et de la fréquence de MARSEILLE comme ville. Pour certains fichiers, la FIFO vers le MC débordera, pour d'autres non. Le mécanisme minimum pour tenir compte de cette situation est de détecter ce débordement et d'avertir l'utilisateur que sa requête est trop complexe pour notre machine.
Ceci n'est pas très satisfaisant, et nous proposons plutot une architecture dont le but est:

1) de traiter les questions simples à la vitesse du disque

2) de traiter les questions complexes à la vitesse du module de calcul

3) de rendre le passage entre les modes 1) et 2) aussi transparent que possible.

Pour cela, on ajoute un tampon de la taille d'une piste (ou plus généralement d'une taille contenant toujours un nombre entier d'enregistrements) à la sortie du controleur du disque.



Le controleur envoie ses données vers le filtre ET vers la mémoire tampon.

Supposons que, lorsque la FIFO Filtre → MC devient pleine, le Filtre se bloque. L'événement catastrophique est donc le débordement de la FIFO d'entrée du Filtre. Quand ceci se produit, on va CONTINUER d'écrire les données courantes dans la mémoire tampon. Lorsque la mémoire tampon est pleine, on reprend l'alimentation du filtre au point où l'on avait interrompue , à partir de la mémoire, et non plus du disque. Quand la mémoire tampon est vidée, on relance la lecture de la piste suivante sur le disque.

Si la FIFO du Filtre ne déborde jamais, tout se passe comme avant, on enchaine les lectures disque. La mémoire tampon est

constamment écrite mais jamais lue.

Il faut noter que cette solution ne revient pas à rallonger la file d'attente, et qu'elle n'est pas comparable à celle consistant à avoir deux buffers alternativement lus et écrits. Ici, si la FIFO déborde, il y a rupture du mode de filtrage vers un mode de rangement pur et simple.

Ainsi, on traite toutes les requêtes de filtrage et de calcul, quelle que soit leur lenteur, de manière totalement transparente. L'hypothèse faite par une telle architecture est évidemment que la file d'attente débordera rarement.

A ce point, il faut se poser deux question importante:

1)- que perd-on à ne plus suivre le débit du disque ?

2)- que gagne-t-on à utiliser des modules spécialisés tels que filtre et MC à partir du moment où on ne travaille plus au vol, mais à partir d'une mémoire tampon ? L'utilisation d'une machine universelle ne serait-elle pas équivalente ?

La réponse à la première question est que l'on ne perd pas grand chose à ne pas suivre en permanence le débit du disque. Si l'on suppose que l'on a à balayer une suite de pistes consécutives, il faudra au pire attendre une demi rotation pour commencer à explorer la piste suivante, soit une augmentation de 50 pour cent. De plus, si une piste est divisée en n secteurs et si l'ordre des enregistrements à l'intérieur d'une piste est indifférent, on ne perd plus que 50 pour cent divisé par n, à condition de pouvoir commencer la lecture au premier secteur trouvé.

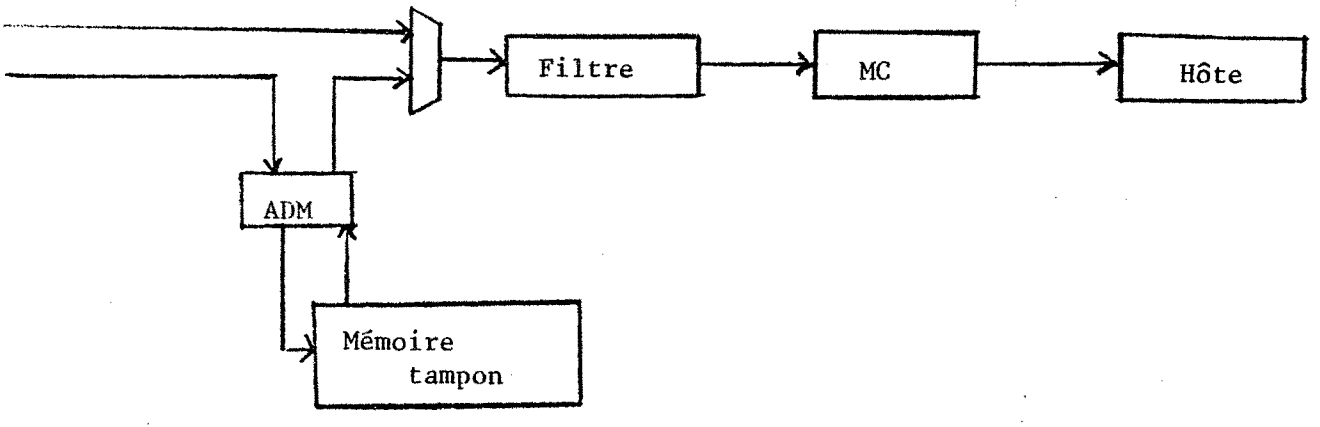
N.B. Dans ce cas, il faut parler de données d'ACCES séquentiel, mais dont l'ORDRE n'importe pas.

Il n'est donc pas catastrophique de ''décrocher'' de temps en temps. (Ceci n'est plus vrai si on travaille sur des données diffusées qui ne passeront qu'une seule fois.)

Si le fait de ne pas suivre le disque ne fait pas perdre beaucoup de temps, cela signifie que le fait de le suivre n'en fait pas gagner beaucoup ...

Dans ces conditions, et en réponse à la deuxième question, l'avantage d'une architecture de type Filtre + MC par rapport à une architecture universelle réside dans:

- la structure pipeline des traitements, et le parallélisme qu'elle introduit; les quatre stations sont:
  * la mémoire tampon et son accès direct mémoire
  * le filtre
  * le module de calcul
  * le système hote



Chacune de ces stations est spécialisée de façon à faire un travail UTILE sur un caractère de données en quelques centaines de nanosecondes. Ce travail se fait en parallèle dans chaque station, et à l'intérieur de chaque station, un parallélisme important existe également:

- entre l'accès à la mémoire et la gestion des registres de l'ADM dans la première station

- parce que les instructions du filtre et du MC sont larges, donc commandant plusieurs actions en parallèle

- parce que les mémoires de données et d'instructions du MC sont accédées en parallèle

Si l'on programme l'ensemble de ces opérations SEQUENTIELLEMENT sur un ordinateur UNIVERSEL, il faudra remplacer chaque action élémentaire exécutée en parallèle dans le pipeline par une ou PLUSIEURS instructions ''standard'' MICROPROGRAMMEES sur une micromachine UNIVERSELLE , et accédant une mémoire UNIQUE, pour les instructions et les données.

Le gain dû au fait de suivre le débit du disque au vol est donc négligeable par rapport à celui dû à la spécialisation de l'architecture. Plus précisément, il en est simplement une retombée: on travaille beaucoup plus vite qu'une machine universelle, et EVENTUELLEMENT assez vite pour suivre le disque.

L'architecture proposée ci-dessus avec une mémoire tampon à la sortie du disque prend donc tout son sens: c'est une architecture pipeline spécialisée qui travaille le plus vite possible, et suit automatiquement le débit du disque si elle le peut.

4 5 RELATIONS ENTRE FILTRE ET MODULE DE CALCUL : VERS LA DISPARITION DU FILTRE

Partant de la fonction de filtrage, nous lui avons d'abord adjoint des fonctions sémantiques simples comme le calcul booléen. Les traitements plus compliqués ont ensuite décentralisés dans un module de calcul, et effectués à la demande du filtre.

Nous étudions ici différents types de couples (Filtre, MC).

Nous en sommes arrivés au point où:

- le filtre ne fait que du filtrage, à l'exclusion de toute action sémantique
- le MC est une machine programmable classique, suffisamment rapide, dont seul le séquencement est relativement spécialisé (pouvoir charger la valeur suivante du compteur ordinal depuis une file d'attente externe, et se bloquer quand elle est vide)

Pour tirer le meilleur parti de cette situation, il faut s'efforcer de:

- simplifier le filtre au maximum

- avoir un MC aussi universel que possible

4 5 1 Simplification des instructions du filtre

En 2, nous avons vu que des largeurs de mots typiques pour les instructions du filtre sont 32 bits (cas de la dichotomie compilée) ou 16 bits (cas des automates tabulés). Ces chiffres ne concernent que la partie arborescente de l'automate, la partie linéaire pouvant être dans tous les cas rangée dans une mémoire de caractères.

Il faut rajouter à ces mots les informations nécessaires pour donner du travail au MC. La solution consistant à rallonger tous les mots est beaucoup trop couteuse car cette rallonge sera inopérante la plupart du temps. Deux autres solutions sont envisageables:

- on crée un nouveau type de mot qui, lorsqu'il est atteint par l'automate, ne sert pas à son séquencement mais est envoyé vers le MC. L'automate saute ensuite ce mot et accède le suivant. Il suffit donc de rajouter un bit à chaque mot pour indiquer si il est ''normal'' ou destiné au MC. Cette solution ajoute un accès mémoire à chaque envoi vers le MC

- une autre solution dérive de la constatation que l'information minimale, mais suffisante, que le filtre doit envoyer au MC est son ETAT courant, (j'ai reconnu tel mot, j'ai atteint la fin de l'enregistrement).

Et cet ETAT est codé sans ambiguité par l'ADRESSE courante dans la mémoire de l'automate. Il suffit donc de marquer (d'un bit) les adresses correspondant à un état significatif et de les

envoyer au MC lorsqu'elles sont atteintes par l'automate. Cette solution est ultra simple et peu couteuse pour le filtre, mais reporte le travail sur le MC qui va devoir associer à chaque adresse de filtre reçue une de ses propres adresses de programme. Ce qui est typiquement un travail de filtrage ... Mais en pratique ces adresses seront peu nombreuses (de l'ordre de la dizaine) et arriveront peu fréquemment (seulement quand un élément significatif est reconnu) Leur reconnaissance peut se faire par Hcodage ou par dichotomie.

N.B.
Dans tous les cas, il faut également commander l'envoi éventuel vers le MC du dernier caractère lu par le filtre. Ceci nécessite un bit supplémentaire par mot de l'automate: on envoie CE vers le filtre si et seulement si le mot courant a ce bit positionné.

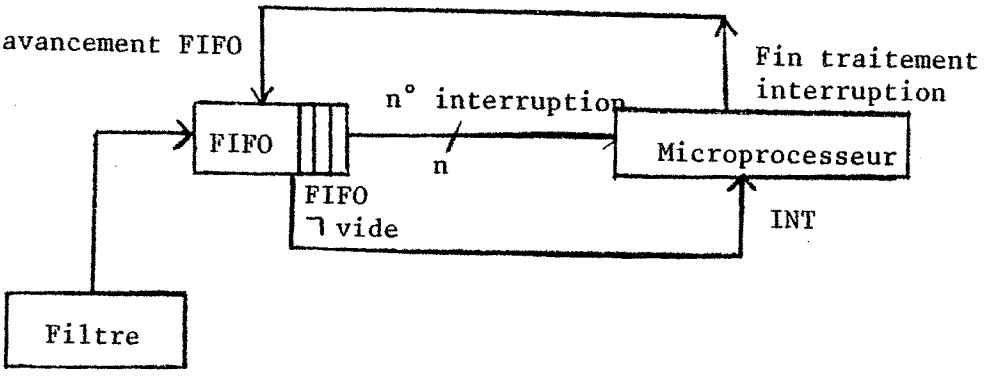
4 5 2 L'interface Filtre → MC vue par le module de calcul

SI l'on vise plus les performances que l'économie, l'interface doit permettre de traiter un caractère reçu en UNE SEULE instruction:
- l'adresse de cette instruction est prise dans 8 bits de la FIFO
- en même temps, les 8 autres bits sont rangés dans le registre DATA
- l'instruction elle-même peut être de la forme MEMOIRE ← MEMOIRE + DATA

Si l'on ne veut pas redéfinir un processeur spécialisé, comment accélérer au maximum la prise en compte du travail envoyé par le filtre, dans le cas où le MC est une machine existante, un microprocesseur par exemple?

1) la suggestion d'utiliser le mécanisme d'interruption du processeur nous a été faite [45]

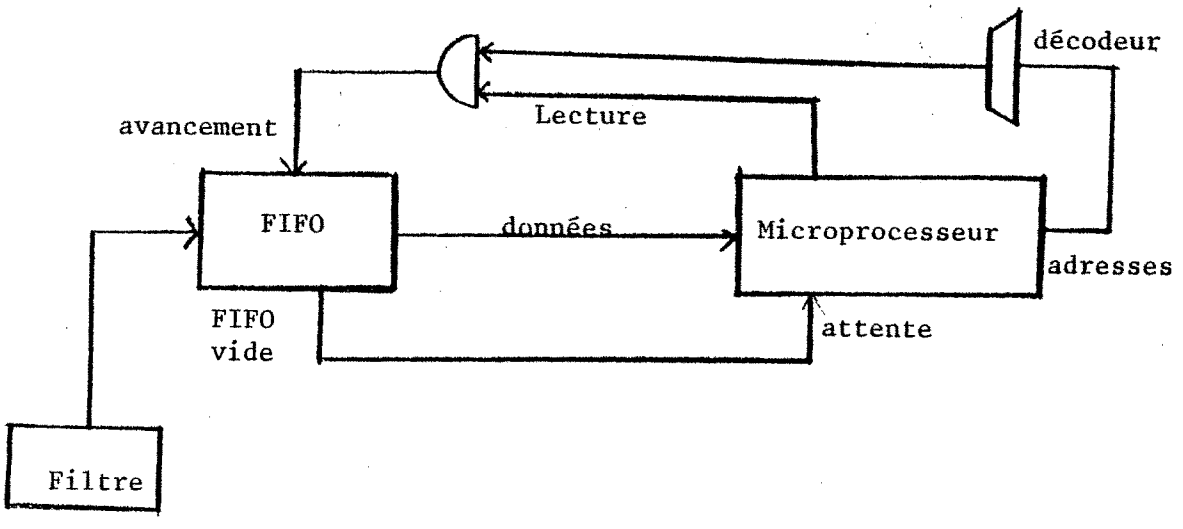
Il faut que le processeur dispose d'un mécanisme d'interruption vectorisé permettant d'en distinguer rapidement un assez grand nombre (16 au moins par exemple). Alors le filtre envoie dans la FIFO une suite de numéros d'interruptions. Le signal d'interruption du processeur est connecté à l'indicateur ''FIFO non vide'' et celui de prise en compte de l'interruption à la commande de lecture de la FIFO.



2) une autre solution est de mettre la FIFO dans l'espace d'instructions du processeur.

Toute demande de lecture mémoire entre deux adresses A0 et A1 va provoquer une lecture du mot suivant dans la FIFO, et c'est ce mot qui sera délivré sur le bus de données. Si la FIFO est vide, son signal FIFO vide est utilisé pour provoquer l'attente du séquenceur du microprocesseur.

Les informations envoyées par le filtre ne sont pas autre chose qu'une suite d'instructions du processeur, destinée à être exécutée séquentiellement.



Lorsque le microprocesseur exècute une instruction de branchement à l'adresse A0, alors il va exècuter en sèquence toutes les instructions envoyèes dans la file d'attente. Il faut veiller à ce que, lors de cette exècution sèquentielle, le compteur ordinal ne dèpasse pas la valeur A1. Une solution simple est que ce flot d'instructions contienne de temps en temps un branchement à A0.

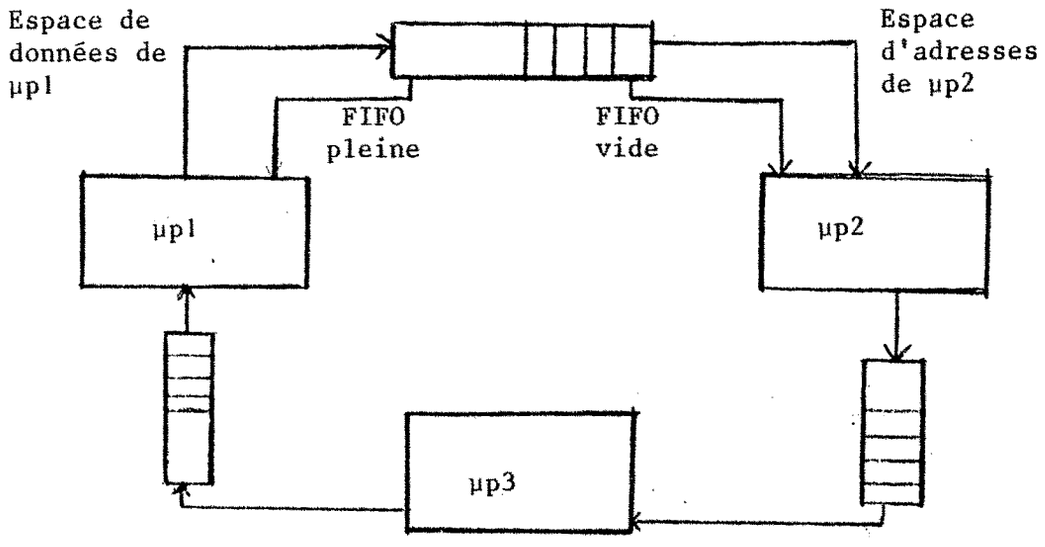
Les instructions lues dans la file d'attente peuvent elles-mêmes provoquer des branchements vers d'autres parties de l'espace d'adressage. Elles peuvent par exemple être des appels de sous-programmes dont le retour s'effectuera sans problèmes par la lecture de l'instruction suivante dans la FIFO.

Chaque fois que le processeur veut continuer son travail par l'exècution des instructions de la FIFO, il lui suffit de se brancher à une adresse comprise entre A0 et A1.

Cette solution nous semble être la plus rapide pour forcer un processeur à exècuter un ensemble d'instructions calculè dynamiquement.

Par exemple, pour rèaliser le calcul boolèen et simuler le positionnement d'une bascule, il suffit d'envoyer une instruction faisant le ou d'un registre avec un masque passè en valeur immèdiate. Une telle instruction s'exècute en moins d'une microseconde sur les microprocesseurs actuels.

Cette technique a certainement une portèe plus gènèrale, et peut être utilisèe pour communiquer des programmes et des donnèes dans un rèseau de microprocesseurs. TOUS les problèmes de synchronisation (gestion de file d'attente, bloquage des processeurs quand les files sont vides ou pleines) sont rèsolus par des boitiers existants d'une manière totalement transparente pour le logiciel, qui se contente de faire des lectures et èritures.



En conclusion, un microprocesseur peut jouer le role du MC de manière satisfaisante dans beaucoup de cas, en particulier en informatique documentaire; il suffit de prévoir une interface matèrielle adèquate entre le filtre et le microprocesseur.

4 5 3 INTEGRATION DU FILTRE AU MODULE DE CALCUL

La conclusion du paragraphe prècèdent est que le module de calcul peut se ''rèduire'' à un microprocesseur si il reçoit surtout des calculs ponctuels à faire (boolèen, comptage), comme c'est le cas en informatique documentaire où le plus gros du travail est fait par le filtre.

Par contre, avec des fichiers informatiques habituels, le module de calcul va recevoir non seulement quelques ordres de calcul ponctuel, mais aussi une partie importante des donnèes, pour faire des cumuls, des statistiques, des projections conditionnelles. Le MC va devoir être beaucoup plus rapide q'un microprocesseur. Il peut être rèalisè sous la forme d'une micromachine universelle, à mots de commande assez larges, et disposant d'une grande mèmoire de donnèes.

Une telle machine (cf le MC de TREFLE) posséde plusieurs modes d'adressage des donnèes, plusieurs modes de calcul de l'adresse de l'instruction suivante. Si on la compare à un filtre, une remarque saute aux yeux:

1) le MC est réalisé en même technologie rapide que le filtre

2) le MC est plus complexe que le filtre

3) cela ne coute presque rien de rajouter au MC l'UNIQUE instruction de séquencement du filtre. Ce qui supprime évidemment tous les problèmes de communication entre filtre et MC ...

L'ensemble filtre + module de calcul est remplacé par une micromachine rapide dont UNE instruction accomplit la fonction de filtrage. Cette instruction peut par exemple être de la forme ''recherche dichotomique':

comparer le caractère en entrée à une constante et choisir entre trois adresses suivantes

Il faut noter que les modifications à apporter au MC sont absolument minimes. Finalement:

MC + ε > MC + Filtre

CHAPITRE 5

TECHNIQUES POUR RESOUDRE LES PROBLEMES SYNTAXIQUES DU FILTRAGE

5 TECHNIQUES POUR RESOUDRE LES PROBLEMES SYNTAXIQUES DU FILTRAGE

Ce chapitre concerne la prise en compte de la STRUCTURE des données à traiter. Nous avons d'abord étudié les problèmes lexicaux, c'est à dire la reconnaissance d'objets terminaux dans les données (mots, chiffres). Puis les problèmes sémantiques (calculs booléens ou arithmétiques) ont été abordés. Jusqu'à présent, il a été fait abstraction de la structure des données. Nous avons toujours considéré l'enregistrement comme une suite de caractères parmi laquelle des mots-clès étaient recherchés. Dans la pratique, un enregistrement a une structure, c'est à dire qu'il est composé de différentes parties que l'on peut désigner au moyen d'un langage particulier.

Par exemple, un enregistrement d'une base de données relationnelle sera constitué d'un ensemble de champs, appelés aussi domaines, chacun de ces champs ayant un nom et un type:

(NOM: texte, AGE: entier , SALAIRE: réel, ADRESSE: texte)

Un enregistrement d'une base de données textuelle pourra décrire un livre et aura la structure suivante:

(TITRE: texte, MOTS-CLES: suite de mots, DATE: suite de nombres, CONTENU: suite de CHAPITRES, un CHAPITRE est une suite de PARAGRAPHES imbriqués)

Avec de telles structures, on peut poser des questions du genre:

- moyenne des AGES des personnes dont l'ADRESSE contient le mot ''avenue''

- TITRES des livres dont la DATE est > 1958 et contenant les mots ''énergie '' et ''Bretagne'' dans le même CHAPITRE.

- TITRES des livres contenant , dans un même CHAPITRE, un PARAGRAPHE contenant les mots ''Cuba'' et ''Miami'' et un PARAGRAPHE contenant les mots ''Sucre'' et ''Coca-cola''

Par rapport aux chapitres 3 et 4 les choses se compliquent: tout traitement lexical ou sémantique se situe dans un contexte structurel donné. Avant de commencer une recherche lexicale, il faut trouver le champ ou la sous-structure dans laquelle la recherche doit être faite. Un calcul booléen peut porter non seulement sur des événements lexicaux (reconnaissance d'un mot) mais aussi sur des phénomènes syntaxiques: tel chapitre contient ou non des sous-chapitres.

Les problèmes posés par la prise en compte de ces structures sont très proches de ceux de l'ANALYSE SYNTAXIQUE telle qu'elle est pratiquée en compilation des langages de programmation. C'est une des raisons pour lesquelles il semble intéressant de considérer les fichiers à analyser comme un langage, et les requêtes comme des GRAMMAIRES, voire des TRADUCTEURS [30].

Cette approche s'est déjà révélée meilleure que les autres pour les traitements lexicaux, et son intérêt va se généraliser au domaine syntaxique. En effet, reconnaitre le début d'un champ est très similaire à reconnaitre un mot-clé, et un automate d'états finis convient parfaitement pour cela. Ainsi, c'est LE MEME MATERIEL qui va servir à reconnaitre les structures et à analyser leur contenu.

Dans toutes les autres architectures de machines bases de données (même dans AFP , qui utilise aussi les automates pour reconnaitre les mots-clés), la structure des données est analysée par un matériel spécifique et trop peu général. Si les champs sont délimités par des caractères spéciaux, ces caractères sont de longueur fixe, donc le nombre de champs est limité. Avec un automate, n'importe quelle séquence de caractères peut être considérée comme un délimiteur.

Pour accepter des structures de données récursives, il suffit de passer de la notion d'automate d'états finis à celle d'automate à pile, ou, plus prosaiquement, de rajouter un mécanisme d'appel de sous-programme avec une pile d'adresses de retour. Ceci va très loin, et permet:

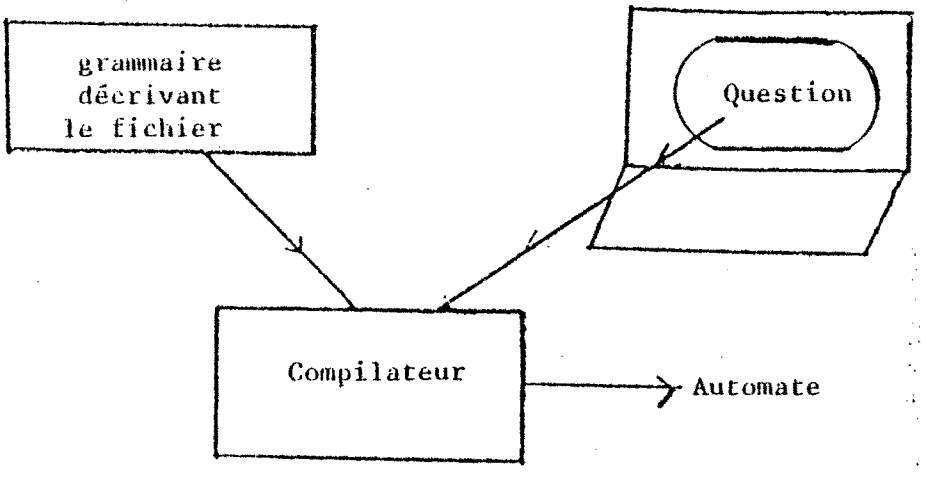
- de décrire la structure des données avec des expressions

régulières (reconnues par des automates d'états finis) ou des grammaires à contexte libre (reconnues par des automates à pile)

- d'accepter aussi bien des organisations de données ''plates'' comme les n-uples de l'algèbre relationnelle, que des organisations hiérarchiques du style SOCRATE ou similaires à celle proposée dans VERSO [46]

Dans toutes les autres machines, la structure de données est figée et CABLEE avec du matériel supplémentaire. Ici, tout reste programmable, sur un matériel plus simple et aussi efficace.

Une conséquence , pour le logiciel d'utilisation d'une telle machine, est que l'automate de filtrage doit être compilé non seulement à partir de la question de l'utilisateur, mais aussi à partir de la description de la grammaire à laquelle obéissent les données. Ceci sera détaillé dans le chapitre consacré à la compilation.



5 1 Reconnaissance de la structure des enregistrements à l'aide d'automates d'états finis

La structure est décrite à l'aide d'un expression régulière permettant de décrire:

- des successions: a b c d

- des alternatives: a | b | c | d

- des répétitions: a *

Exemple: ((a b | c | d) * (x y z) | d) *

En pratique, les lettres a, b, c représentent pour nous des sous-structures terminales de l'enregistrement, et les expressions étoilées des sous-structures non terminales répétitives.

On peut décrire de cette manière des structures plates:

Fichier :: (A texte1 B texte2 C texte3 D texte4 E) * F

les texte1 à 4 sont les valeurs des champs proprement dites, A B C D indiquent à la fois la fin de n'importe quel champ et le début d'un champ précis. E termine n'importe quel champ et indique la fin d'un enregistrement et le début du suivant. F indique la fin du fichier.

Avec une telle représentation, l'ordre des champs n'a pas d'importance, et un champ peut être répété. Le nombre de champs n'est pas borné par le nombre de caractères délimiteurs ni par autre chose puisque l'indicateur d'un champ peut être aussi long que l'on veut: A123456A par exemple. Les champs sont évidemment de longueur variable, sans qu'il soit nécessaire de préciser leur longueur.

Ces facilités ne coutent rien, puisque l'analyse de la structure va être faite au vol. Les systèmes logiciels classiques ne peuvent se permettre une structure aussi souple, et sont obligés d'utiliser une description plus rigide des champs, permettant leur localisation rapide en mémoire aléatoire: formats fixes, ordre imposé, format variable avec la longueur du champ codée de manière fixe (souvent sur plus d'un octet, donc prenant plus de place qu'un caractère réservé), liste des adresses de

début des champs en tête de l'enregistrement.

Les machines base de données n'utilisant pas les automates pour la reconnaissance des structures imposent aussi des contraintes rigides comme celle consistant à avoir des délimiteurs de taille FIXE.

A la limite, rien ne nous empêche de mettre EN CLAIR le NOM DU CHAMP comme délimiteur, précédé d'un unique caractère spécial:

(/NOM texte /AGE texte /SALAIRE texte .....)

Ceci simplifie la saisie et l'édition des données, ainsi que leur transfert d'une base à une autre. C'est un facteur supplémentaire d'INDEPENDANCE DES DONNEES.

L'analyse par états finis permet aussi de représenter et d'analyser des structures hiérarchiques comme celles de SOCRATE, ou celles de langages de programmation (COBOL, PL/1, PASCAL).

Exemple:

```
1 VILLE  10 FOIS
   2 NOM 20 CARACTERES
   2 QUARTIER    20 FOIS
     3 NOM 20 CARACTERES
     3 POPULATION  ENTIER
     3 RUE       30 FOIS
       4 NOM 20 CARACTERES
       4 IMMEUBLES  ENTIER
     3 FIN
   2 FIN
1 FIN
```

Dans tous ces langages, on doit déclarer la taille maximale de tous les éléments de la structure, et la mémoire correspondante est allouée statiquement ou dynamiquement (dans le cas de Socrate).

Ici, il suffit quasiment de reprendre telle quelle la structure déclarée pour pouvoir l'analyser avec un automate d'états finis:

/1VILLE /2NOM paris /2QUARTIER /3NOM passy /3POP 10000 /3RUE 4NOM blanche /4IMM 500 /3RUE /4NOM grande /4IMM 500 /2QUARTIER 3NOM clichy /3POP 5678 /3RUE /4NOM foch ....

Dans le projet VERSO [46] sont proposés des algorithmes pour gérer de telles structures au vol.

5 2 Reconnaissance de structures à l'aide d'automates à piles

Si l'on ajoute au jeu d'instructions du filtre la possibilité de faire des appels récursifs de sous-programmes, on obtient la puissance de reconnaissance des automates à piles. On peut alors reconnaitre des structures d'enregistrements récursives, parenthésées.

N.B. Dans le cas précédent, la DESCRIPTION de la structure est parenthésée, mais la structure elle-même ne l'est pas.

En pratique, cette possibilité trouve peu d'applications pour les structures des bases de données existantes où les structures ne sont pas récursives. Elle est utile en informatique documentaire où un paragraphe contient d'autres paragraphes. Elle permet aussi d'analyser au vol des structures de données dynamiques comme celles du langage LISP.

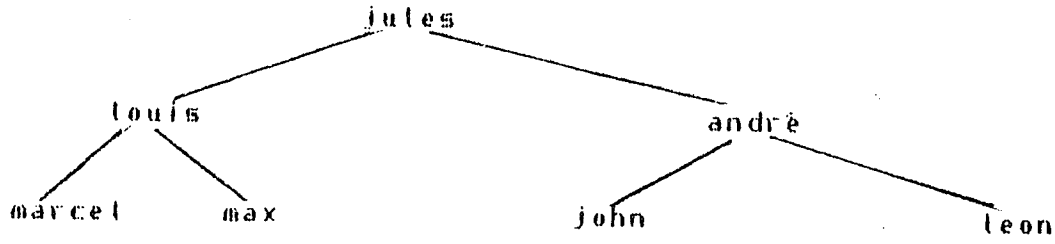
Considérons par exemple un arbre généalogique ''à l'envers'', c'est à dire l'arbre des descendants d'une personne donnée. Cette structure ne peut être décrite en SOCRATE ou PASCAL -autrement qu'avec des pointeurs- car son nombre de niveaux n'est pas borné.

Descendants :: (NOM →liste de fils)
Fils :: NOM ou descendants

Exemple:

(jules → (louis → marcel,max),(andré → john,leon))

Ce qui correspond à l'arbre suivant



Une différence avec les arbres finis précédents est que l'on peut remplacer LOCALEMENT la chaine de caractères ''marcel'' par ''(marcel → xxx, yyy, zzz) sans créer de nouveaux séparateurs autres que les parenthèses. Ici les séparateurs sont INTERPRETES alors qu'ils sont COMPILES avec les arbres finis.

5 3 Imbrication des hiérarchies syntaxiques et sémantiques.

Les opérations sur des fichiers ayant une structure hiérarchique (finie ou non) sont plus complexes que sur des structures plates comme les n-uples des bases relationnelles ou des fichiers classiques. En effet l'unité de sélection de base, l'enregistrement, n'existe plus. Un fichier n'est plus une suite d'enregistrements indépendents les uns des autres, mais une énorme expression régulière ou une phrase d'un langage hors-contexte.

Si on cherche les quartiers de PARIS dont au moins deux rues ont plus de 100 immeubles, (dans l'exemple de structure du paragraphe précédent), il va falloir

- savoir si on est ou non dans une sous-structure VILLE de nom PARIS

- mémoriser tous les noms de quartiers jusqu'à la fin de leur sous-structure

- faire un calcul à chaque nombre d'immeubles par rue

- à la fin d'un quartier, garder ou non le nom de ce quartier si la condition sur les rues est satisfaite.

Il a été montré que la gestion de la mémoire des champs en attente d'être sélectionnés (comme les quartiers ici) peut être faite simplement à l'aide d'une pile moyennant certaines restrictions raisonnables sur la structure des enregistrements [46].

L'avantage principal de cette organisation des données est de FACTORISER au maximum les valeurs alors que dans une base relationnelle , on devrait avoir, pour chaque quartier un enregistrement (ville, quartier) et pour chaque rue un enregistrement (rue, quartier). La taille totale de la base est donc diminuée. De plus, beaucoup d'opérations qui nécessitent une jonction avec des fichiers classiques (donner le nom des villes où une rue s'appelle Foch par exemple) se font ici par un simple balayage du fichier hiérarchique.

Cependant, si l'on possède un mécanisme pour effectuer les jonctions au vol, il n'est pas clair que la taille de la mémoire balayée dans l'unique fichier structuré soit inférieure à celle balayée dans tous les fichiers plats, d'autant plus qu'un fichier plat est plus facile à indexer qu'un fichier hiérarchisé.

Cette solution diminue la taille STATIQUE des fichiers, mais pas la taille DYNAMIQUE du balayage, qui conditionne le temps de réponse.

Son apport essentiel est de permettre de traiter au vol des structures hiérarchiques, qui demandent un nombre considérable d'accès disque sur les systèmes classiques du fait de leur implémentation par POINTEURS des relations de hiérarchie.

5 4 Evaluations sémantiques sur des données à structure hiérarchique

Soit la structure:

```
1 ARTICLE
  2 AUTEUR
    3 NOM
    3 AGE
  2 TITRE
  2 REFERENCE
     3 LIEU
     3 DATE
  2 TEXTE
```

```
    2 FIN
1 FIN
```

et la question:

Trouver les articles satisfaisant la condition C←((C1 et C2) ou (C3 et C4)) et non (C5 et C6), avec

C1 ← une condition sur le TEXTE : contient MAXI ou MICRO
C2 ← une condition sur le TITRE : contient KNUTH ou SMITH
C3 ← une condition sur la REFERENCE : la date est > 1970
etc ...

C est une fonction de 6 variables, C1 à C6, qui sont elles-mêmes des conditions booléennes pouvant porter sur plusieurs variables. En tout, C peut avoir 15 ou 20 variables élémentaires. Si l'on applique la solution de tabulation exposée en 3, la taille de la table de vérité risque d'exploser.

Une meilleure solution est de décomposer ce calcul en deux niveaux:

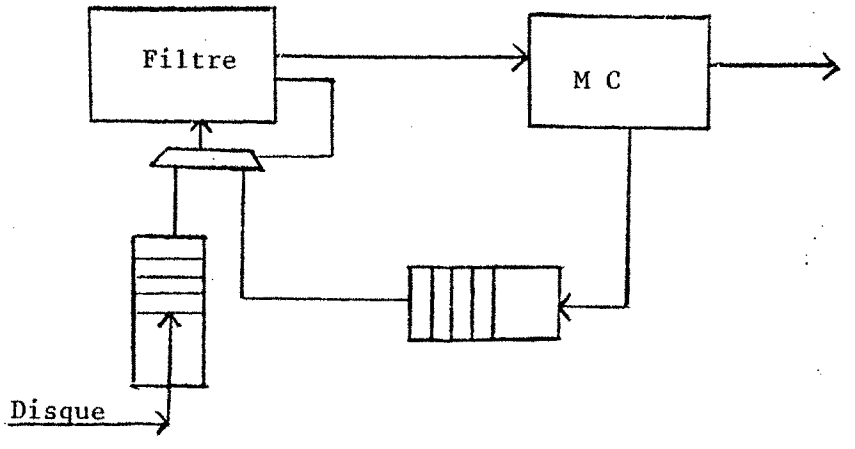
- un calcul individuel de chaque Ci dans 5 tables séparées
- un calcul final de C avec une autre table, indicée par les résultats des tables précédentes.

On peut alors évaluer des expressions booléennes énormes, pourvu qu'elles soient hiérarchisables. Une condition favorable pour cela est que l'enregistrement soit lui-même hiérarchisé (comme le montre l'exemple précédent) et que le langage d'interrogation respecte cette structuration.

Une généralisation intéressante est de permettre à de tels calculs partiels et intermédiaires d'influencer non seulement la fonction sémantique finale, mais aussi les fonctions lexicales et syntaxiques .

Une manière très élégante de réaliser ceci est de pouvoir réinjecter un résultat de calcul intermédiaire dans le flot de caractères analysé par les automates du filtre.

Dans le cas où le filtre est associé à un module de calcul, cela implique que le module de calcul puisse renvoyer des résultats vers le filtre, et que celui-ci puisse au choix lire le caractère suivant à analyser depuis la file de sortie du disque ou depuis celle du MC.



Si le filtre est intégré au MC, la communication d'un résultat de calcul à l'activité de filtrage est évidemment triviale.

Une solution est de permettre à un sous-programme de rendre comme résultat un caractère, et que ce caractère soit immédiatement pris comme le caractère courant CE analysé par le filtre.

Voici un exemple montrant l'intérêt de ce mécanisme:

Supposons que les mots-clés sont divisés en diverses catégories (noms propres, dates, lieux, etc). On pose une question de la forme:

(auteur contient des mots des catègories 1 ou 3 et pas 5) et (titre contient les catègories 4 et 9) et (texte ne contient pas les catègories 3 et 11).

Il suffit d'écrire un sous-programme qui reconnait les mots-clès et se termine à la fin de chaque mot par le retour d'un caractère donnant la catègorie du mot reconnu. Un tel programme ne fait pas autre chose que l'analyse lexicale habituelle des compilateurs qui transmet à l'analyse syntaxique le codage de chaque identificateur.

Le sous-programme qui analyse la partie ''auteur'' peut donc appeler le sous-programme lexique, et, lorsqu'il reprend le contrôle et lit le caractère CE suivant, il recevra en fait le rèsultat du programme lexique. Selon la catègorie reçue, il positionnera ou non une bascule pour prèparer l'èvaluation de sa condition booléenne privée. A la fin de la partie auteur, le sous-programme auteur peut évaluer sa condition booléenne, et RETOURNER son rèsultat à un programme appelant qui recevra également de la même manière les rèsultats des conditions pour le titre et le texte.

Ceci suppose donc qu'un sous-programme puisse retourner non seulement un caractère constant, mais aussi le rèsultat d'une condition booléenne privée.

Finalement, on a une hièrarchie de sous-programmes calquès sur la structure syntaxique de l'enregistrement et qui peuvent à tout moment choisir de lire le CE suivant depuis l'entrée du filtre ou depuis le rèsultat retournè par le sous-programme qu'ils viennent d'appeler.

A chaque niveau, il faut disposer d'une logique privée d'évaluation des conditions booléennes (vecteur de bascules et table de vérité) dont les commandes de remise à zéro et d'évaluation doivent être individualisées.

5 5 Des sous-programmes aux PROCESSUS de filtrage

Une extension naturelle est de généraliser les sous-programmes en PROCESSUS, ou COROUTINES, pour leur permettre de conserver leur contexte syntaxique (adresse courante) ou sémantique (bascules) entre deux appels successifs.

Remarquons que ceci est inutile si la structure des calculs est strictement arborescente comme dans l'exemple précédent, où le mécanisme de pile suffit.

Voici un exemple d'utilisation des coroutines:

On doit analyser une suite d'enregistrements comportant entre

autres une date composée de champs jour, mois et année. On désire vérifier si les enregistrements sont bien classés par dates successives. Il suffit de construire trois coroutines, chacune possédant une liste de jours, mois et années (liste cyclique pour les jours et mois). A chaque appel, ces dernières coroutines retournent l'une des informations suivantes:

'1' si le jour (mois,année) courant est bien le successeur du précédent

'2' si le jour (mois) courant est bien le successeur du précédent, et l'on est repassé au début de la liste

'3' si la séquence n'est pas respectée

'4' si le jour (mois,année) courant est égal au précédent

L'enregistrement courant est correct si et seulement si les réponses des coroutines jour,mois,an sont (dans l'ordre):

- 1 4 4

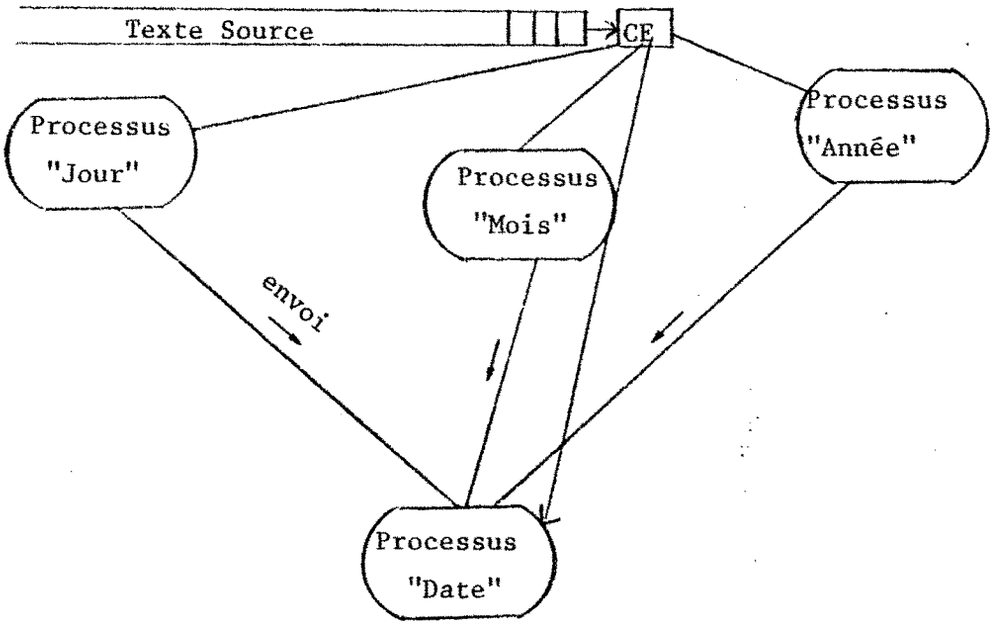
ou

- 2 1 4

ou

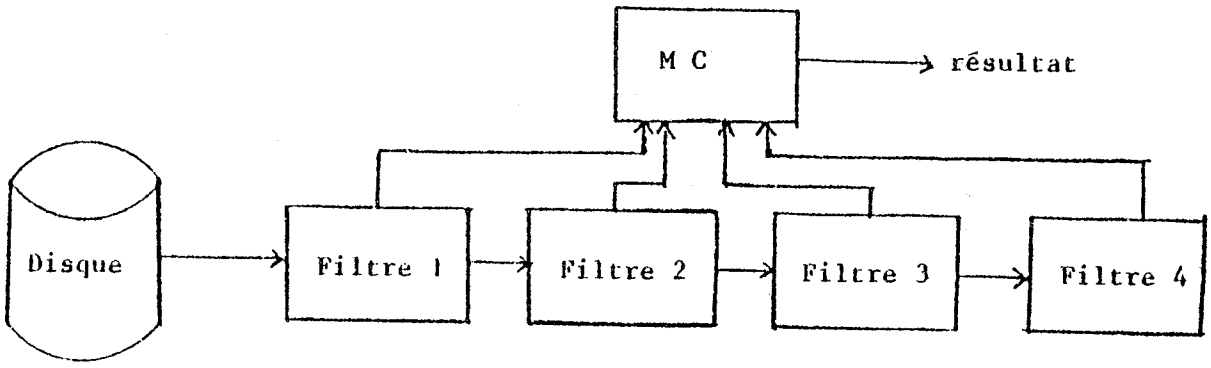
- 2 2 1

ce qui est facilement reconnaissable par un automate au niveau appelant.



L'introduction d'un tel mècanisme de coroutines (ou processus) apporte des possibilitès multipliées pour l'analyse des donnèes au vol.

Une alternative purement matèrielle est de le rèaliser PHYSIQUEMENT en montant en sèrie plusieurs filtres, chacun envoyant des caractères dans la sortie du suivant:



L'inconvénient est que le nombre de processus et leurs liens étaient figés (un processus par filtre) et surtout que le prix d'une telle configuration est prohibitif et sa puissance inutile, puisque seul le premier filtre reçoit un fort débit, comme cela a été vérifié par une modélisation des performances [48].

Une fois encore, la solution programmée s'avère meilleure que la solution câblée, sans conséquences pour les performances utiles.

5.6 Conclusion sur la partie syntaxique

En conclusion, les besoins dus à l'analyse des structures de données confortent notre analogie avec les mécanismes de compilation.

L'analyse des structures par des automates d'états finis, des automates à piles ou des réseaux d'automates pseudo-parallèles est simple à réaliser matériellement, et assez rapide pour travailler au vol avec des disques. Elle apporte une puissance et une souplesse incomparables aux solutions câblées des autres machines bases de données. La structure des données analysée est totalement PROGRAMMABLE au moment de la compilation de la requête, et s'adapte aux structures classiques plates (relations) ou hiérarchiques, aussi bien qu'à des structures plus complexes (textes, listes).

Le dernier point abordé (réseaux d'automates pseudo-parallèles) apparente l'architecture que nous proposons à des machines et des langages connus dans la littérature sous le nom de ''DATA FLOW MACHINES'' ou ''DATA-STREAM LANGUAGES'' [49] [50]. Nous croyons beaucoup au pouvoir d'expression et à la cohérence de tels systèmes, et espérons y trouver un point de départ pour apporter une solution au problème crucial déjà énoncé: comment programmer les applications pour pouvoir répartir automatiquement le calcul entre un traitement au vol et un traitement classique, condition nécesaire pour que les machines bases de données tiennent leurs promesses sur le terrain.

CHAPITRE 6

APPLICATIONS DES FILTRES A LA GESTION DES FICHIERS ET DES BASES DE DONNEES

6 APPLICATION DES FILTRES A LA GESTION DES FICHIERS ET DES BASES DE DONNEES

Ce chapitre a pour but de montrer que les filtres peuvent jouer de multiples roles dans l'implèmentation de systémes de gestion de donnèes sur disque, que ces donnèes soient organisèes de maniére très rudimentaire sous forme de fichiers ou sous la forme de bases de donnèes complètes.

Nous verrons que les filtres permettent dans tous les cas d'obtenir de meilleurs temps de rèponse que les systémes logiciels classiques, et que ces gains sont d'autant plus importants que les problémes à rèsoudre sont complexes. Dans certains cas, on verra en particulier que le temps de rèponse est indèpendant de facteurs comme:

- la complexitè de la requête
- le nombre d'utilisateurs simultanès
- le nombre de critéres d'accès

Dans ce chapitre, nous considèrons des enregistrements très formattès, comme ceux que l'on trouve habituellement dans les applications d'informatique de gestion. Rappelons que les filtres conviennent èglement remarquablement bien au traitement de donnèes peu formattèes comme les textes de la langue naturelle.

Un des aspects importants de ce travail consiste à montrer que les filtres sont un mècanisme UNIVERSEL de gestion des donnèes, dans la mesure où ils remplissent un certain nombre de conditions qui ont ètè prècisèes dans les chapitres prècèdents et en particulier:

- une vitesse de parcours suffisante pour suivre le dèbit des donnèes

- la possibilitè de faire des comparaisons $<$ et $>$ et pas seulement $=$ et $\neq$

- la possibilitè d'èvaluer au vol des conditions booléennes

Le rèsultat est une architecture beaucoup plus universelle qu'aucune autre construite ou proposèe à ce jour. Elle traite en effet aussi bien:

- les structures textuelles ET les structures informatiques

- les chaines de caractères ET les nombres

- les structures des enregistrements ET les valeurs dans ces structures

Pour garder le maximum de gènèralitè aux paragraphes qui suivent, nous suppposerons l'utilisation d'une architecture minimale, se composant de:

- un disque et son controleur

- un filtre avec calcul booléen

- un processeur universel et sa mèmoire (typiquement un microprocesseur)

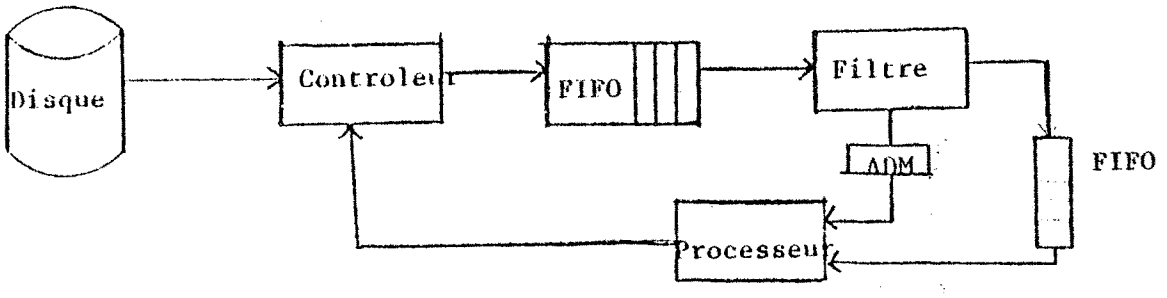
Les liaisons entre ces constituants sont les suivantes:

- les donnèes sont envoyèes octet par octet dans la file d'entrèe du filtre par le controleur du disque

- le processeur commande les opèrations de lecture et écriture de l'ensemble disque-controleur.

- le filtre peut sortir par une FIFO soit une valeur immédiate, soit un résultat d'une condition booléenne, soit le dernier caractère qu'il vient de lire en entrée.

Cette FIFO peut être lue par programme par le processeur général



Ceci concernait l'architecture matérielle utilisée. Dans le même souci de simplicité et de généralité nous utiliserons un format d'automate simplifié dans nos exemples.

- tout l'automate sera représenté en arborescence (donc sans parties linéaires)

- l'instruction de base est celle du format dichotomique compilé

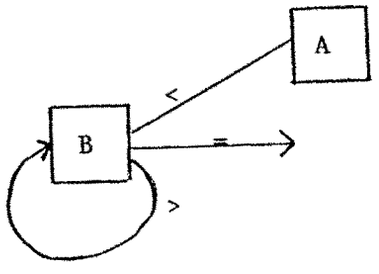
VC	A=	A<	A>

- la règle d'avancement dans le texte d'entrée va être généralisée, pour permettre les retours arrières:

* actuellement, cette règle est d'avancer si et seulement si on va à l'adresse A= (succès de la comparaison entre CE et VC)

* si l'on ATTEND un caractère donné, il faut pouvoir avancer aussi dans les deux autres cas.

* en fait, il n'y a pas symétrie entre A< et A>, par exemple si l'on attend un caractère parmi deux possibles, comme dans:



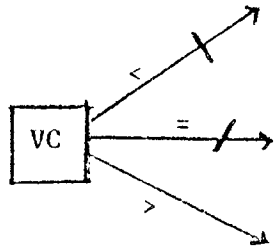
si CE est inférieur à B, il ne faut pas lire le CE suivant, mais il faut le lire si CE est supérieur.

Il faut donc un bit commandant l'avancement pour le cas < et un autre pour le cas > .

* il peut aussi être utile de NE PAS avancer en cas d'égalité; c'est le cas où un sous-programme se termine par une comparaison avec égalité et que le programme appelant aimerait lui aussi connaitre la valeur du dernier caractère lu par le sous-programme.

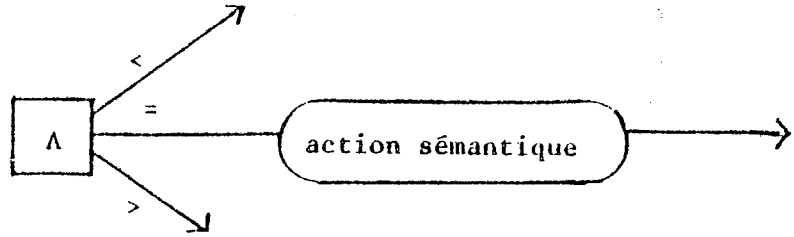
En conclusion, nous proposons tout simplement d'avoir, associé à chacune des trois adresses, un bit qui dit si il faut avancer ou non dans le texte lorsque le choix s'est porté sur cette adresse. Tous les cas ci-dessus sont alors spécifiables.

La représentation graphique de notre instruction sera:



avec la convention qu'une flèche est barrée si et seulement si il faut lire un nouveau CE.

De plus, d'éventuelles actions sémantiques pourront être exécutées. Elles seront représentées graphiquement ainsi:



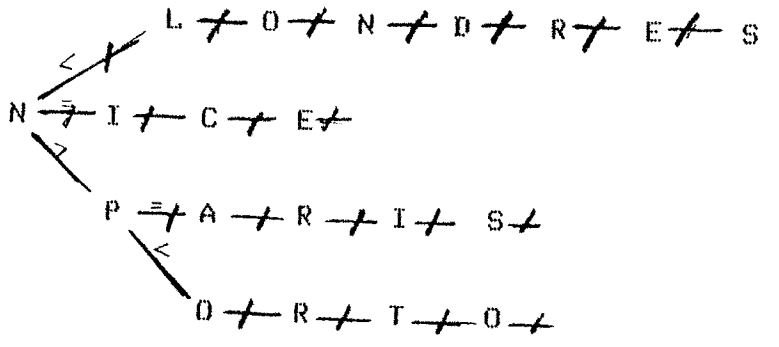
Nous allons maintenant étudier les applications des F.S. des plus simples aux plus compliquées, c'est à dire des enregistrements aux bases de données, en passant par les différentes organisations de fichiers.

6 1 UTILISATION DES F.S. AU NIVEAU D'UN ENREGISTREMENT

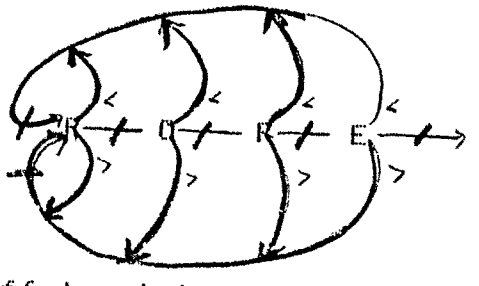
6 1 1 RECHERCHE D'UNE SOUS-CHAINE DANS UNE CHAINE

Pour savoir si un champ contient N'IMPORTE OU une ou plusieurs sous-chaines données, il suffit de construire l'automate qui reconnait ces sous-chaines et de reboucler toutes les sorties d'échec vers le début. Exemple:

PARIS ou NICE ou LONDRES ou PORTO



Mais en général les choses ne sont pas aussi simples. Si l'on cherche le mot POPE, l'automate suivant est incorrect:



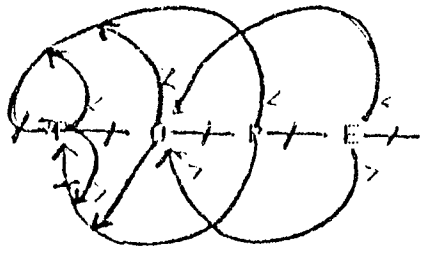
En effet, si le texte analysé est de la forme:

.....POPOPE.....

il contient bien POPE mais l'automate ne s'en aperçoit pas.

Après avoir reconnu POP, il attend un E et trouve un O, considère que c'est un échec et et reprend au début avec la recherche d'un P, recherche qui échouera ainsi que les suivantes.

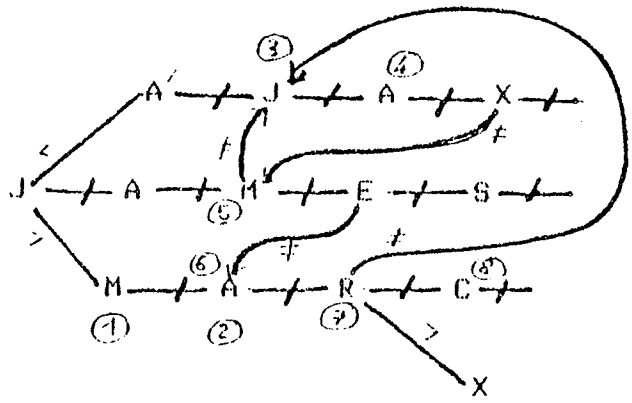
L'erreur est de ne pas s'apercevoir que les dernières lettres rencontrées avant l'échec peuvent être les les lettres de début du mot recherché. La comparaison aurait donc du se poursuivre non au début du mot, mais plus loin. Le bon automate est le suivant:



Quand on ne trouve pas E, il faut continuer par comparer à O SANS AVANCER (flèches non barrées).

Ce problème existe aussi dans le cas de plusieurs chaines:

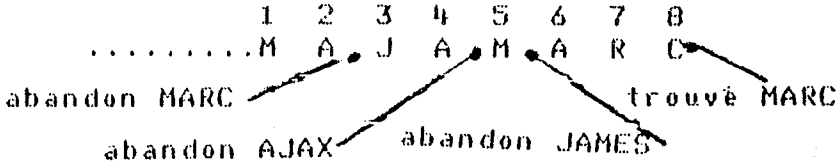
Exemple: MAX ou MARC ou AJAX ou JAMES



N.B. Seules les flèches de retour non triviales ont été dessinées.

Exemple:

Si le texte source est:



Un algorithme pour calculer le bon automate est donnè par KNUTH [51]. Il est utilisè dans le compilateur prèsentè plus loin.

Cet algorithme est rappelè en annexe A2

.6 1 2 Rappel sur l'èvaluation au vol des conditions boolèennes.

Rappelons brièvement la technique dejà prèsentèe pour èvaluer les conditions boolèennes.

Chaque fois que l'automate atteint un ètat correspondant à une condition boolèenne atomique (un mot-clè est trouvè, par exemple), il positionne une bascule à 1. A la fin de l'enregistrement, l'ensemble des bascules est utilisè comme adresse pour lire une table de bits qui contient la table de vèritè de la fonction boolèenne à èvaluer.

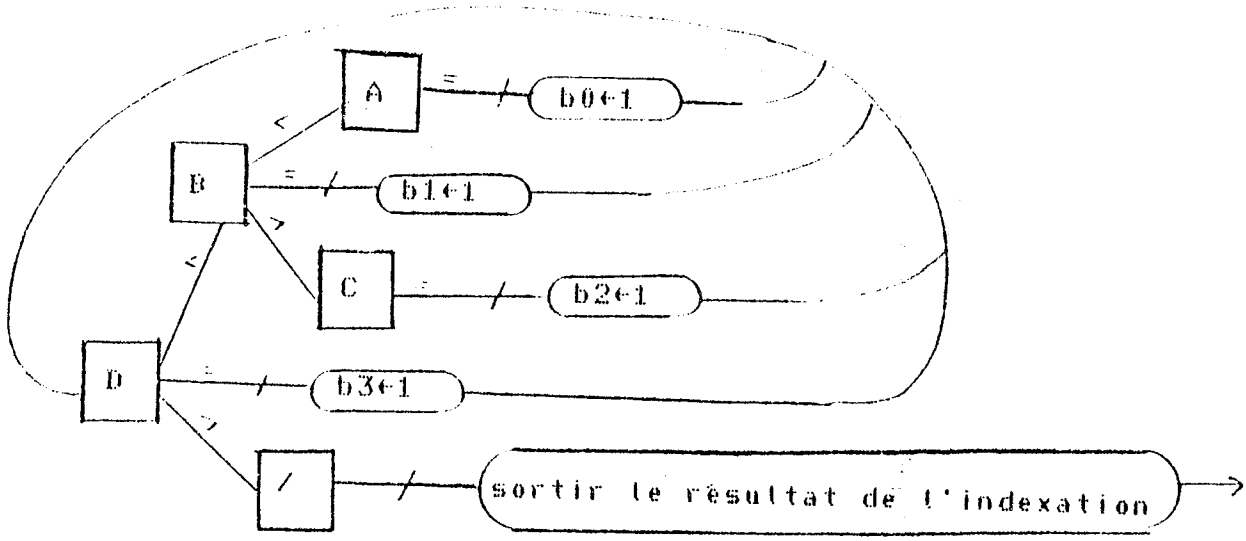
Exemple:
Trouver les enregistrements contenant (A et B) ou (C et non D). L'enregistrement se termine par ''/''.

On utilise 4 bascules:

- b0 assΦcièe à la reconnaissance de A
- b1 à celle de B
- b2 à celle de C
- b3 à celle de D

Les bascules sont remises à zèro au dèbut de chaque enregistrement.

L'automate est:



La conséquence fondamentale des techniques utilisées est que:

- la vitesse d'exécution de telles questions est indépendante de leur complexité (nombre de mots-clés et d'opérateurs booléens)

- une seule passe des données suffit pour exécuter la question

6 1 3
RELATIONS D'ORDRE ET COMPARAISONS ENTRE CHAINES ET NOMBRES.

Les exemples précédents portaient sur la recherche d'EGALITE de chaines de caractères. C'est l'activité la plus fréquente en informatique documentaire. En informatique classique, on manipule aussi des NOMBRES, et il faut pouvoir leur appliquer des comparaisons < et > et pas seulement = et ≠.

N.B. Même en informatique documentaire, il faut pouvoir dire des choses du genre: 'Nombre de pages inférieur à 100' ou 'Date

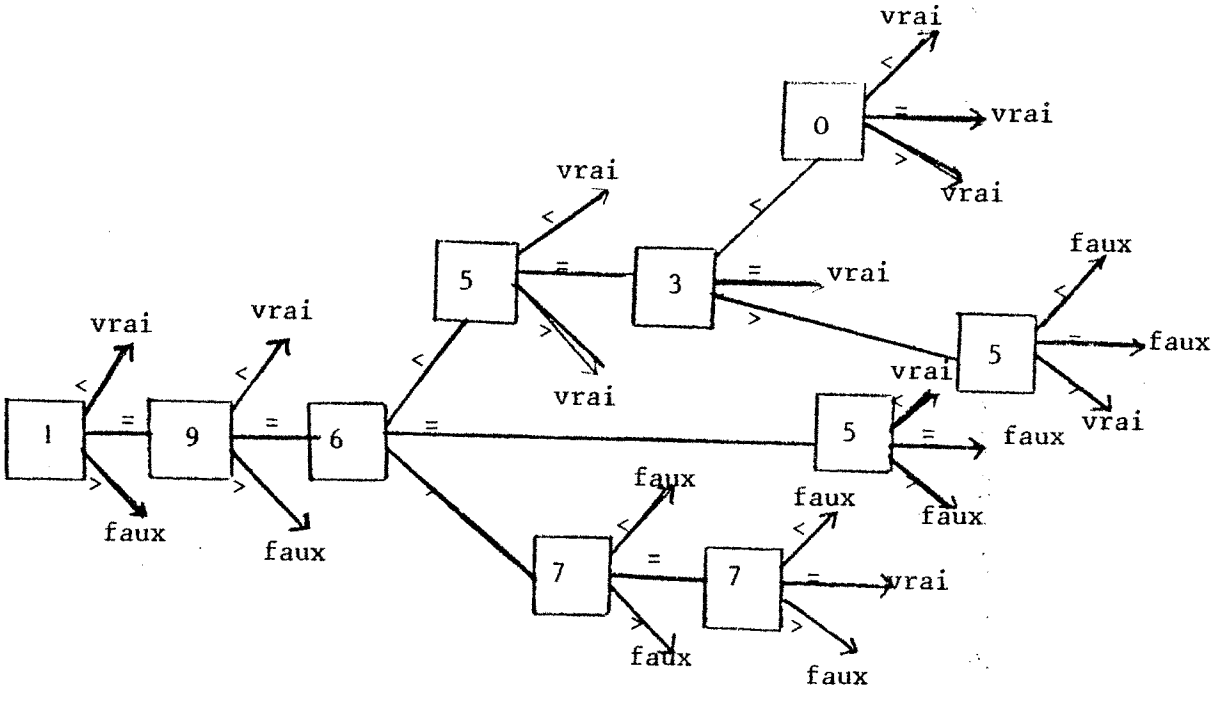
comprise entre 1950 et 1960'.

De telles comparaisons sont très faciles à faire avec nos automates où chaque comparaison distingue TROIS sorties =, <, et >, contrairement à ceux de la machine AFP [ ], qui a seulement les sorties = et ≠.

Ainsi l'expression:

Date = 1953 ou 1977 ou comprise entre 1955 et 1965 ou < 1950

peut être évaluée par l'automate suivant:



on peut vérifier par exemple que:

1250 → vrai
1960 → vrai
1967 → faux

Les sorties par ''VRAI'' peuvent positionner une bascule qui

sera utilisée dans un calcul booléen de niveau supérieur.
On voit que l'automate sert à lui seul non seulement à faire des comparaisons, mais aussi à évaluer une condition booléenne.

Un seul filtre peut donc accepter DEUX niveaux de calcul booléens de la forme:

(Age ≤ 17 ou ≥ 48) et (Salaire compris entre 300 et 400)

Cas des nombres de format variable.

Pour pouvoir comparer des nombres de format fixe, il suffit de recevoir les poids forts en premier.
Si les nombres sont en format variable (virgule flottante pour le calcul scientifique ou nombre de chiffres variables pour la gestion), on peut adopter un format du type suivant:

Un nombre est représenté par:

- son signe, avec le '+' codé supérieur au '-', par exemple respectivement 1 et 0

- la position de son chiffre le plus significatif. Cette position est un nombre positif ou nul , codé sur un nombre FIXE de caractères (2 dans notre exemple)

- les chiffres représentant la valeur absolue du nombre, les zéros à droite étant supprimés. De plus, si le nombre est négatif, les chiffres sont remplacées par leur complément à la base (10 dans notre exemple)

Exemple, en supposant que la position la moins significative est 10 puissance moins 10.

34 → +1134
123 → +12123
1234.56 → +13123456
-0,00467 → -07643
0.0000000001 → +001 (le plus petit nombre positif

représentable)

Avec de telles notations, le filtre peut traiter des conditions de la forme:

(température > 456.008 ou > 32,67)

N.B. La nécessité de présenter les poids forts en tête va compliquer l'exécution d'éventuelles additions série.

6 1 4 TRAITEMENT DE PLUSIEURS QUESTIONS SUR LE MEME FLOT DE DONNEES

L'idée est la suivante: puisque le temps de traitement est indépendant de la complexité de la question, pourquoi ne pas fusionner en un seul programme plusieurs questions si l'on dispose de suffisamment de ressources mémoire (mémoire d'instructions et d'évaluation booléenne) ?. Le seul problème est de pouvoir distinguer à la fin les différentes réponses individuelles dans la sortie du filtre.

Cette technique a été proposée initialement par les auteurs de la machine AFP [26].

- on accepte jusqu'à N questions, et on associe une table de vérité et un ensemble de bascules à chacune d'elles

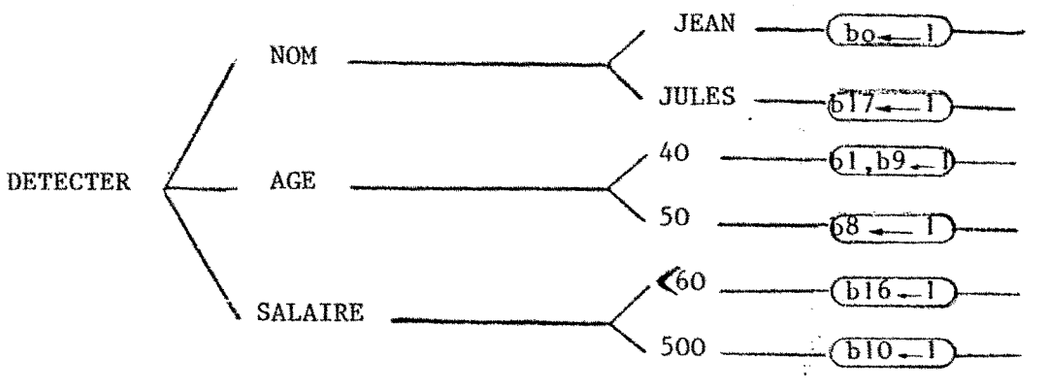
- on fusionne les automates de reconnaissance des mots-clés de chacune des questions en un seul

- à la fin, le filtre sort les N résultats booléens, et le processeur sait à quelles questions l'enregistrement courant a satisfait.

Exemple:

Q1 : (Nom = JEAN) et (Age = 50)
Q2 : (Age = 40 ou 50) et (Salaire = 500)
Q3 : (Salaire < 60) et (Nom = JULES)

Le programme final aura l'allure suivante (en résumant)



Q1 a sa table de vérité adressée par B0 ... B7
Q2 a sa table de vérité adressée par B8 ... B15
Q3 a sa table de vérité adressée par B15 ... B23

Cette technique est intéressante dans un environnement multiutilisateurs où certains fichiers sont fréquemment interrogés. La contrainte est d'attendre un certain temps l'accumulation des questions avant de les exécuter. Cette technique améliore le débit global du système à pleine charge plutot que le temps de réponse à une question individuelle.

Les fichiers interrogés doivent être évidemment parcourus séquentiellement. Il peut s'agir de fichiers de données mais aussi d'index.

Avec cette technique, la taille maximum de fichier pour laquelle une recherche séquentielle 'naive' avec un FS est toujours meilleure que n'importe quel algorithme logiciel sophistiqué devient grande. Exemple:

En UNE seconde, on peut traiter N questions très complexes sur un fichier de 1 MILLION de caractères.

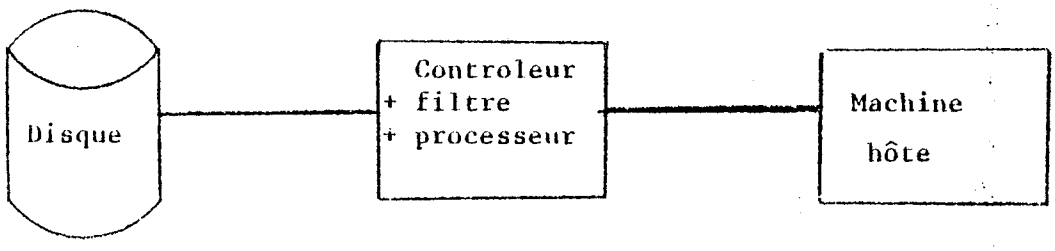
Bien que cette technique soit difficile à appliquer en toute généralité dans un système de bases de données, elle semble très intéressante dans des cas très spécifiques: fichiers petits, beaucoup d'utilisateurs, forte charge.

6 2 UTILISATION DES FS POUR LES OPERATIONS PORTANT SUR LES FICHIERS

La partie précédente examinait comment le filtre décidait si un ENREGISTREMENT donné satisfaisait ou non une condition donnée. En pratique, les enregistrements sont organisés en fichiers, un fichier contenant des enregistrements ayant tous la même structure, de telle sorte que le même programme de filtrage peut être appliqué à tous les enregistrements. Les performances des FS suggèrent qu'un accès séquentiel à tout le fichier peut être acceptable si celui-ci est assez petit: on peut traiter au vol environ un million de caractères par seconde.

Le FS va donc examiner TOUS les enregistrements pour les noter chacun par '0' ou '1'.

Vu d'un ordinateur hote classique, tout se passe comme si c'était le périphérique lui-meme qui avait résolu la question, puisqu'il reçoit dans sa mémoire centrale directement les bons enregistrements.



Avec un système classique, les enregistrementd doivent d'abord être amenés en mémoire centrale, où ils sont filtrés par du logiciel. Même si ce logiciel est très sophistiqué (c'est à

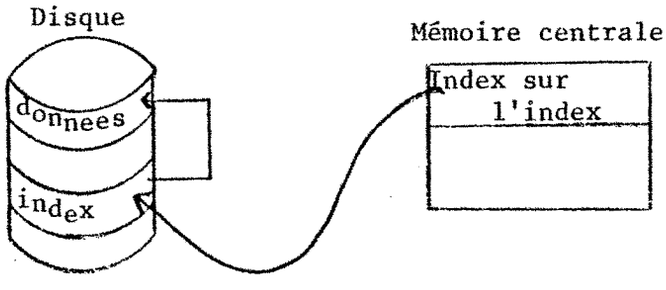
dire très gros et complexe), la solution FS est TOUJOURS PLUS RAPIDE , puisque le travail est fait durant le transport des donnèes, AVANT même que le logiciel ait pu commencer son travail!

Le temps de rèponse est pour nous ègal à le taille du fichier multilpliè par la vitesse de transfert du disque. Si le fichier fait 100 millions d'octets, le temps de rèponse est de 100 secondes. Et puisque ce temps est indèpendant de la complexitè de la question, il est aussi ègal à 100 secondes pour une question aussi simple que:

'Trouver l'age de l'employè numèro 456'

Et heureusement, les systèmes logiciels actuels ont un temps de rèponse infèrieur à la seconde pour de telles questions ...

Leur recette est d'augmenter la complexitè du logiciel et des structures de donnèes. Par exemple, le fichier sera triè selon les numèros d'employè, et il existera un second fichier, dit fichier index qui, pour chaque bloc (= sous-ensemble physique du fichier) indiquera la plus petite valeur du numèro qu'il contient. De plus, ce fichier indexest lui aussi indexèpar une table d'index suffisamment petite pour tenir en mèmoire centrale.



En consèquence, trouver l'age de l'employè 456 ne prendra que deux accès disques , plus un peu de temps de calcul,soit au plus 200 à 300 millisecondes, plus de 300 fois plus rapide que notre solution matèrielle!

Il est èvident que, même si les dèsavantages de notre solution dècroissent rapidement quand la question se complique, une

solution purement sèquentielle n'est acceptable que pour des fichiers suffisamment petits , de l'ordre de 500 000 caractères. (ce chiffre pouvant être multipliè par le nombre de questions qui sont fusionnèes en un même automate).

Les fichiers plus gros nècessitent des accès plus sophistiquès (du genre index) MEME avec des FS.
Le paragraphe qui suit montre prècisèment que les FS sont AUSSI trés intèressants pour exploiter les index.

6 2 1 FICHIERS A UNE SEULE CLE D'ACCES

C'est le cas dans l'exemple:

'trouver l'age de l'employè numèro 456'

où le numèro est supposè identifier un employè de manière unique.

Nous supposons que le fichier est divisè en BLOCS, qui peuvent etre un ensemble de secteurs, pistes ou cylindres... Le fichier est triè selon le numèro d'employè.
On crèèe un FICHIER INDEX de la manière suivante:

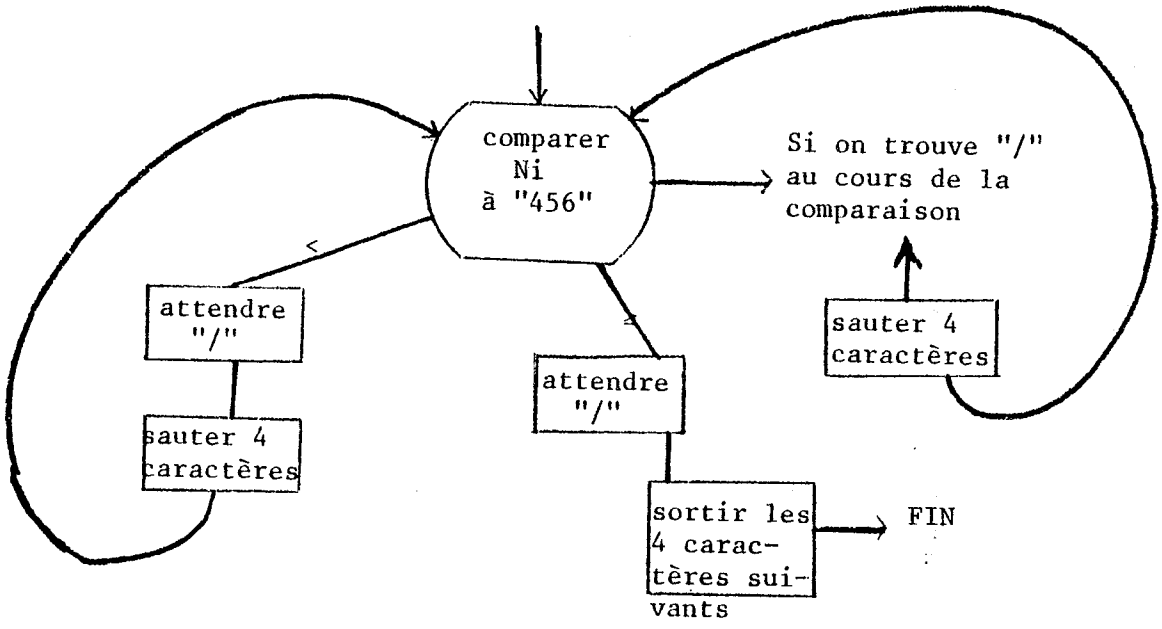
N0,B0,N1,B0,N2,B1,N3,B2, ..., Nn,Bn-1, ...

où Ni est la plus petite valeur des numèros d'employè dans le bloc Bi
La reprèsentation de ces entitès est la suivante:

- Bi a un format fixe: 4 nombres dècimaux par exemple

- Ni est de longueur variable et se termine par un caractère rèservè, '\' par exemple

Alors le filtre peut être programmè ainsi:



Dès qu'un numéro plus grand que 456 est trouvé, les quatre caractères suivants représentant l'identificateur du bloc précédent sont envoyés dans la FIFO de sortie. Le processeur attend en permanence cette information, et peut commander le positionnement du disque sur ce bloc dès la fin du secteur courant.

Le filtre est alors traversé par les enregistrements du bloc qui contient un numéro égal à 456, s'il en existe. Le bon enregistrement est alors trouvé en explorant séquentiellement ce bloc avec la question:

Age de employé dont numéro = 456

Finalement, si l'on compare avec les performances d'un système logiciel classique, les performances des FSDS sont les suivantes, en supposant l'utilisation d'un disque type CDC 9766 [52]

- un accès disque au fichier index: 30 ms
- une demi (en moyenne) traversée d'un bloc d'index sans accès direct mémoire (ADM): 8 ms
- un accès disque au fichier données: Soit A la durée de cet accès.
- une demi (en moyenne) traversée d'un bloc de données avec un ADM pour le seul bon enregistrement: 8 ms

La solution logicielle nécessite:

- un accès disque au fichier index: 30 ms
- un transfert de tout un bloc d'index en mémoire centrale, central par ADM: 16 ms
- du temps de calcul pour explorer le bloc d'index; Soit B la durée de ce calcul.
- un accès disque à un bloc de données: Soit A la durée de cet accès.
- un transfert de tout un bloc de données en mémoire centrale: 16 ms
- du temps de calcul pour trouver le bon enregistrement: soit B la durée de ce calcul.

Le filtre travaille pendant (30 + 8 + A + 8) = 46 + A.

Le logiciel travaille pendant (30 + 16 + B + A + 16 + B) = (62 + A + 2×B).

Le gain dû au filtre est de (62 + A + 2×B) ÷ (46 + A). Il est d'autant plus important que A est petit et B est grand).

La valeur minimum de A est zéro (l'index est sur le même cylindre que les données), et sa valeur maximum est 30 ms (l'index est n'importe où par rapport aux données).

La valeur minimum de B est proche de zéro (si le logiciel utilise des algorithmes très sophistiqués en mémoire aléatoire) et son maximum peut dépasser 25 ms (si l'algorithme logiciel traite les caractères en série, avec 3 microsecondes par caractère)

Le gain dû au filtre est donc compris entre:

(62 + 50) ÷ 46 et (62 + 30) ÷ (46 + 30)

soit entre 2,43 et 1,21

De plus, le logiciel nècessite un blocage du processeur central pendant 32 ms pour transfèrer les donnèes dans sa mèmoire, soit une diminution de son

utilisation comprise entre 28 et 34 pour cent.
Un calculateur hote utilisant notre architecture minimale reçoit le rèsultat plus rapidement que ne pourrait faire n'importe quelle mèthode logicielle sophistiquèe, et en utilisant des structures de donnèes plus simples: par exemple, les enregistrements à l'intèrieur d'un bloc de donnèes ne SONT PAS TRIES.

Les FS ètant encore un concept très expèrimental, beaucoup d'autre mèthodes pour les utiliser efficacement pourront sans doute être trouvèes. L'exemple proposè avait pout but essentiel d'illustrer les deux avantages principaux:
- les comparaisons de clès et l'analyse de la structure des enregistrements se font 'au vol'
- le système central n'intervient pas durant la recherche

6 2 2 ACCES A UN FICHIER SUR PLUSIEURS CRITERES

Il s'agit de questions du type:
trouver le numèro des personnes dont l'age = 56 et le salaire = 50000
Les solutions logicielles consistent à avoir PLUSIEURS INDEX, un sur l'age, l'autre sur le salaire. Mais le fichier ne peut être trièe sur plusieurs clès à la fois, et la solution prècèdente ne peut donc pas se gènèraliser à plusieurs clès d'accès. Il faut utiliser des FICHIERS INVERSES.

Pour chaque champ Ci utilisable dans une question, il existe un fichier dont les enregistrements sont de la forme:

(Valeur du champ, liste de blocs)

où les blocs de la liste sont ceux qui contiennent au moins

un enregistrement avec cette valeur dans le champ Ci.

Exemples:

- pour le salaire:

(20 000, B1,B5,B6, ...)
(23 400, B1,B2,B4, ...)
(30 000, B4,B6,B60, ...)
etc. ....

- pour l'age:

(35, B1,B5,B9, ...)
(36, B1,B2, ...)
(38, B6,B8, ...)

Ces fichiers inverses peuvent être triés sur les valeurs du champ auxquel ils sont associés.

Pour résoudre notre question, la solution logicielle consiste à interroger le fichier inverse des ages pour trouver l'ensemble des blocs associés à AGE = 56, et à accéder celui des salaires pour trouver la liste des blocs associés à SALAIRE = 50000, puis à faire l'intersection de ces deux listes.
On termine par une recherche séquentielle sur les blocs contenus dans cette intersection.

Il est clair qu'un paramètre important de cette méthode est la taille des blocs:
- si le bloc est petit, la recherche séquentielle finale est courte, mais les fichiers inverses ont beaucoup d'entrées et les calculs ensemblistes sur les listes de blocs sont volumineux
- si les blocs sont gros, la recherche finale est longue, mais les fichiers inverses sont petits, ainsi que les listes à manipuler.

Dans tous les cas, l'usage d'un filtre améliore les performances, puisqu'il exécute mieux les taches suivantes:

- les recherches dans les fichiers inverses, qui sont triés sur une seule clé

- les recherches séquentielles finales

Cependant, il y a mieux à faire que de reprendre les algorithmes et les structures de données issus des solutions logicielles.

Tout d'abord, il faut se rappeler que si le fichier est assez petit, l'accès séquentiel est suffisant et constitue la solution la plus rapide, sans s'embarrasser d'index ou de fichiers inverses.

Si par exemple la question est:

(Salaire = 20 000 ou 30 000 ou 40 000) et (Age = 50 ou 53 ou 57)

alors la solution logicielle peut nécessiter environ dix accès disques (et plus concrétement dix opérations d'entrée sortie dans le superviseur), avec beaucoup de temps de calcul pour gérer les listes. On atteindra vite un temps de 1 seconde.

Si le fichier (sans compter ses inverses évidemment) fait au plus 1 million de caractères, toute cette complexité logicielle est inutile, puisqu'un accès séquentiel par des filtres va aussi vite!

De plus, le temps de réponse du logiciel se dégrade encore si la question comporte non seulement des égalités, mais aussi des ($< \leq \geq > \neq$);

Trouver les gens dont (Age entre (40 ou 50) ou (60 ou 65)) et (salaire $>$ 50 000 ou $\leq$ 20 000)

Dans ce cas, la taille maximun des fichiers pour laquelle la recherche séquentielle est compétitive est bien supérieure au million d'octets, puisque les listes de blocs à manipuler deviennent énormes. Par exemple, 'Salaire $<$ 20 000' peut représenter le tiers des entrées ...

6 2 3 UNE TECHNIQUE D'ACCES MULTICRITERES ADAPTEE AUX FILTRES

Nous proposons maintenant une autre technique pour interroger les fichiers multicritères avec des FS

Soient c1,c2, ... ,cn l'ensemble des champs par lesquels le fichier peut être accédé.

Pour chaque bloc de données bi, contruisons son RESUME de la forme suivante:

(kmin1,kmax1,kmin2,kmax2, ... , kminj,kmaxj,.. .. ,bi)

où kminj est la plus petite valeur du champ cj présente dans le bloc bi, et kmaxj la plus grande.

Plutot que de faire une recherche globale sur tout le fichier, on fait d'abord une recherche sur les RESUMES, et on ne lira pas le bloc bi, si, au vu de son résumé, on a pu conclure qu'il ne contient pas d'enregistrements satisfaisant à la question.

N.B. Les résumés ne sont pas éparpillés sur le disque en tête de chaque bloc, mais regroupés consécutivement en un seul fichier séquentiel. Le parcours des résumés ne cause qu'un seul déplacement de bras.

La question utilisée pour parcourir les résumés est évidemment différente de la question de départ, mais elle s'en déduit très simplement selon les règles suivantes:

$cj = A \rightarrow (kminj \leq A)$ et $(kmaxj \geq A)$
$cj < A \rightarrow (kminj < A)$
$cj > A \rightarrow (kmaxj > A)$
$cj \neq A \rightarrow$ non $((kminj = A)$ et $(kmaxj = A))$

Tout le reste de la question (les conditions booléennes) reste inchangé. Il suffit de faire les remplacements indiqués.

Il n'est pas nécessaire que les kmin et les kmax soient exactement égaux aux minimuns et maximums, il suffit qu'ils soient respectivement plus petits ou plus grands. Cette propriété peut être utilisée pour diminuer leur longueur. Par exemple, kmin = ACAPULCO peut être remplacé par kmin= ACAP.

De plus, une chaine commune au début du min et du max peut être factorisée.

Finalement, le résumé d'un bloc représente un très faible supplément de données. Exemple:

Si un bloc est une piste de disque SMD [52], soit approximativement 16K octets, un résumé sur quatre champs avec une longueur de 5 caractères pour les kmin et kmax, aura une taille de environ 50 octets, soit 0.3 pour cent des données. (Ce serait de 1 pour mille si le bloc faisait trois pistes)

Si on a un très grand fichier de 250 millions d'octets constitué de 2.5 millions d'enregistrements de 100 octets, alors:

- la taille du résumé est de 250 000 caractères, indépendamment du nombre d'enregistrements et de la distribution des valeurs des champs
- le résumé est parcouru en 250 millisecondes, INDEPENDAMMENT DE LA COMPLEXITE DE LA QUESTION ET DE LA TAILLE DU RESULTAT.
- on peut de plus poser plusieurs questions en un seul accès au résumé puisque c'est un fichier séquentiel.

Il est clair qu'une recherche séquentielle sur le résumé donne un résultat similaire à une exploration des fichiers inverses: c'est une liste de blocs à parcourir.

La solution FSDS + RESUMES a un avantage considérable qui est d'avoir un temps de réponse CONSTANT (et court) pour l'obtention de cette liste, lorsque la complexité de la question varie.

6 2 3 1 MISE EN OEUVRE DE LA TECHNIQUE DES RESUMES

Un point à optimiser est la diminution de la taille de le liste finale de blocs.

Si un fichier (numéro d'employé,salaire,age,département) est

trié sur les numéros d'employés, alors une question de la forme:

'trouver les employés avec (age entre 25 et 35) et (salaire entre 300 et 400)

va aboutir à un nombre de blocs énorme,(peut-être tous), puisque les ages et salaires sont distribués aléatoirement parmi les blocs.

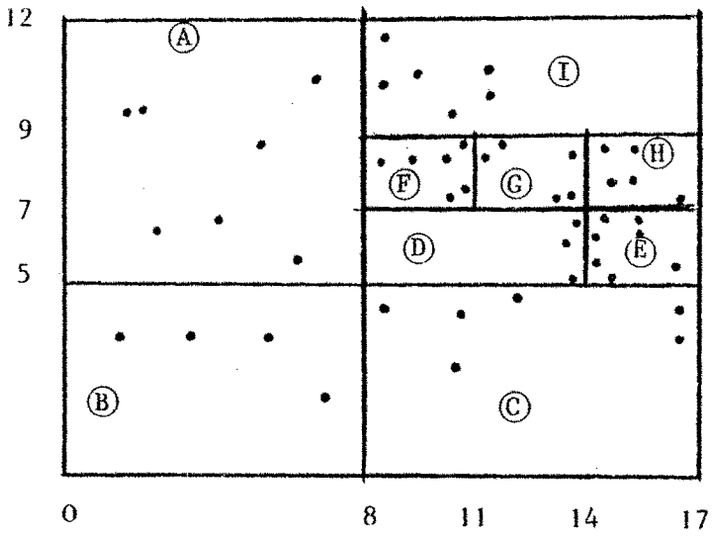
La technique des résumés est bonne SEULEMENT SI, dans CHAQUE résumé, les minimums et les maximums de CHAQUE champ sont suffisamment proches l'un de l'autre. C'est un problème de CLASSIFICATION ('clustering'). Nous voudrions que les données obéissent à une propriété de LOCALITE et de CONTINUITE, c'est à dire que, pour n'importe quelle valeur d'un champ (ou de plusieurs champs), tous les enregistrements ayant la même valeur ou des valeurs proches soient situées dans le même bloc ou dans le plus petit ensemble possible de blocs.

En formalisant un peu le problème, si l'on a N champs, chaque enregistrement occupe un point dans espace à N dimensions, avec les valeurs de ses champs comme coordonnées. On veut que deux points voisins dans l'espace à N dimensions soient aussi voisins sur le disque, et si possible dans le même bloc.

Le problème est: comment appliquer un espace à N dimensions dans un espace à 1 ou 2 dimensions, en préservant la proximité?

Des mathématiciens se sont penchés sur ce problème il y a cent ans déjà [53], et différents auteurs l'ont repris et appliqué dans différents domaines de l'analyse numérique et de l'informatique [54]

Voici un petit exemple à deux dimensions, qui permet de visualiser le problème:



Liste des résumés:

A: (0 - 8) (5 - 12)
B: (0 - 8) (0 - 5)
C: (8 - 17) (0 - 5)
D: (8 - 14) (5 - 7)
E: (14 - 17) (5 - 7)
F:(8 - 11) (7 - 9)
G:(11 - 14) (7 - 9)
H: (14 - 17) (7 - 9)
I: (8 - 17) (9 - 12)

Ici, la taille d'un bloc est limitée à huit points. Il y a 9 blocs, le résumé est constitué de 9 fois deux couples de valeurs.

La solution par fichiers inverses implique 50 entrées pour chacun des deux champs.

Le filtre nécessite 1 accès disque plus le temps de lecture du résumé, et les fichiers inverses nécessitent deux accès plus deux lectures de secteur, et ce pour CHAQUE CLE, plus le travail sur les listes de blocs.

Nous avons vu comment la technique des résumés tirait avantage du filtrage au vol; il faut maintenant examiner la gestion de tels fichiers, c'est à dire l'insertion et la suppression d'un enregistrement.

6 2 3 1 Insertion d'un enregistrement.

Soit un fichier à n critères, numérotés de 1 à n, correspondant aux champs C1, C2, ... Cn.

Il faut ranger un nouvel enregistrement E , constitué des valeurs A1, ... ,An pour chaque champ.

On va le ranger dans le bloc où il se trouverait s'il était déjà dans le fichier. Ce bloc est donc trouvé en posant la question:

(C1 = A1) et (C2 = A2) .... et (Cn = An)

Cette question est transformée comme indiqué précédemment pour parcourir le résumé:

(Kmin1 ≤ A1) et (Kmax1 ≥ A1) .... et (Kminn ≤ An) et (Kmaxn ≥ An)

Pour être certain que cette question sélectionne au moins un bloc, il faut que l'ensemble des blocs constitue un RECOUVREMENT de l'espace à n dimensions considéré.

On obtient alors un ou plusieurs numéros de bloc où E doit être rangé, parmi lesquels on en choisit un.
Si ce bloc choisi est suffisamment vide pour contenir l'enregistrement à insérer, alors l'insertion est faite. Sinon, il va falloir créer un nouveau bloc dans le fichier.

Soit B1 le bloc plein et B2 le nouveau bloc à créer.
L'ensemble des enregistrements de B1 plus le nouvel enregistrement E vont être répartis entre B1 et B2 de la manière suivante:

1)- on choisit un champ i (1 ≤ i ≤ n) qui va servir à faire le partage

2)- on trouve une valeur K telle que les deux ensembles:

* E1 : enregistrements dont Ci ≤ K
* E2 : enregistrements donr Ci ≥ K

aient des tailles aussi proches que possible l'une de l'autre (donc chacune aussi proche que possible de la moitié de la taille maximum d'un bloc)

3)- on range E1 dans B1 et E2 dans B2

4)- dans le résumé de B1, on remplace l'actuelle valeur kmaxi (maximum pour le champ i) par K

5)- le résumé de B2 est identique à l'ancien résumé de B1 où la valeur kmini (minimum pour le champ i) est remplacée par K

Il faut noter que le fait que kmaxi de B1 = Kmini de B2 conserve la propriété de RECOUVREMENT de l'espace par l'ensemble des blocs.

Le point le plus important dans cet algorithme est le choix du champ i qui va servir à faire le partage.

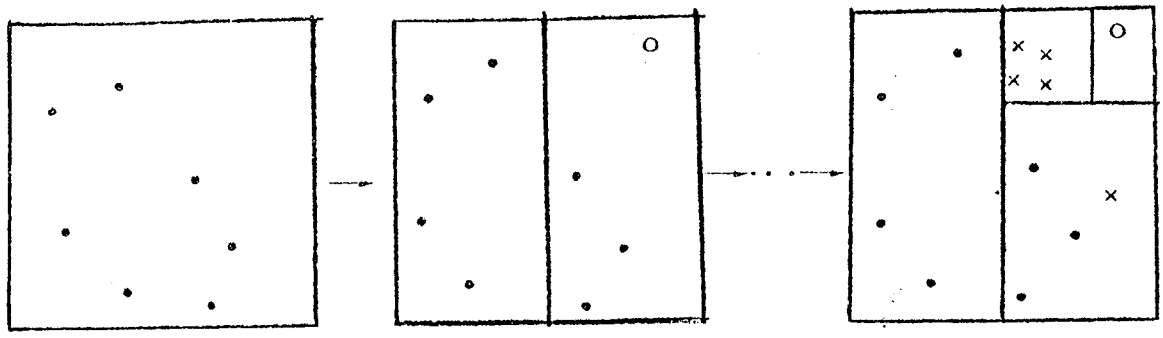
Dans les simulations que nous avons faites, les valeurs des différents champs sont des nombres réels. Un poids Pj est associé à chaque champ j.

La valeur choisie pour le champ qui servira au partage est celle qui minimise l'expression:

$$(Kmax_j - Kmin_j) \div P_j$$

Intuitivement, il s'agit de couper le bloc perpendiculairement à son coté le plus long, la longueur obéissant à une pondération donnée le long de chaque axe. Le but est que tous les blocs aient les mêmes rapports entre leurs longueurs.

Voici un exemple à deux dimensions, avec P1 = P2 = 1



6 2 3 2 Suppression d'un enregistrement

Pour supprimer un enregistrement donnè E = (A1,A2, ..., An), on commence par retrouver le bloc B qui le contient.

Si, aprés suppression de l'enregistrement, la taille du bloc reste supèrieure à un seuil donnè, alors l'opèration est terminèe (on ne modifie pas le rèsumè de ce bloc).

Sinon, le bloc doit être supprimè, et ses enregistrements rèpartis dans d'autres blocs. Appelons a priori ces blocs B1, ,BN.

L'opèration va se faire en deux temps:

1)- Dèterminer les blocs B1, ,BN

2)- Supposer que B ET AUSSI les enregistrements qu'il contient n'existent plus, et modifier en consèquence les rèsumès des Bj, afin de boucher le trou causè par la suppression de B.

3)- réinsérer dans le fichier les enregistrements restants de B

Voici l'algorithme qui permet de déterminer les blocs Bi.

Nous avons besoin de données supplémentaires dans le résumé. Chaque fois qu'un bloc BP (P comme Père est dédoublé pour donner naissance à un bloc BF (F comme Fils), on note:

- dans BF , la coordonnée choisie pour faire la coupure, baptisée Coupure.Père.

- dans BP, la même coordonnée baptisée Coupure.Fils.
Lorsque un noeud n'a pas de fils , sa Coupure.Fils prend la valeur 0.

Si un noeud donne naissance à plusieurs fils, on ne conserve que la valeur de Coupure.Fils la plus récente.

Dans la suite, la notation suivante est utilisée:

KminⒸi,jⒿ est le minimum du champ j dans le bloc i
KmaxⒸi,jⒿ est le maximum du champ j dans le bloc i

Dans ces conditions, si l'on doit supprimer le bloc B, les blocs Bi sont calculés ainsi:

1)- si B a au moins un fils: (Coupure.Fils de B > 0)

k ← Coupure.Fils de B

Un bloc appartient à l'ensemble recherché si et seulement si son résumé satisfait les conditions suivantes:

- kminⒸb,kⒿ = kmaxⒸB,kⒿ

- et, pour toute coordonnée j autre que k:

(kmin[b,j] ≥ kmin[B,j]) et (Kmax[b,j] ≤ kmax[B,j])

Pour chacun des blocs b ainsi trouvés, il faut faire:

kmin[b,k] ← kmin[B,k]

De plus,

Coupure.Père de Fils de B ← Coupure.Père de B

Il faut donc pouvoir retrouver le bloc fils du bloc B. Une solution classique consisterait à maintenir à jour une description complète de l'arbre des relations Père-Fils entre les blocs. Ceci compliquerait beaucoup les structures de données liées aux résumés.

Par chance, les structures existantes peuvent être exploitées par un critère de filtrage pour trouver le dernier fils d'un noeud:

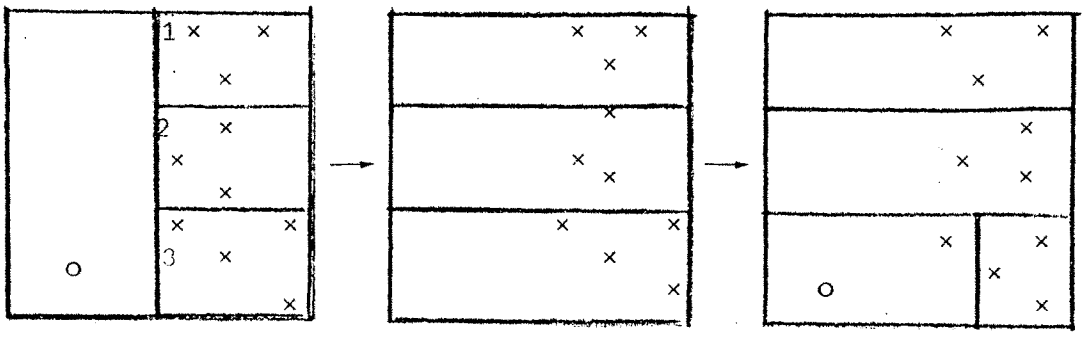
Etant donné un bloc B, soit k = Coupure.Fils de B. Le dernier fils de B est le seul bloc F tel que:

- kmin[F,k] = kmax[B,k]

- et kmin[F,l] = kmin[B,l] pour tout l ≠ k

Exemple:

Taille minimum d'un bloc: 2 points ; taille maximum: 4 points



2)- Si B n'a pas de fils: (Coupure.Fils de B = 0)

k ← Coupure.Père de B

Les blocs recherchés sont les blocs b tels que:

kmax[b,k] = kmin[B,k]

et, pour toute coordonnée j différente de k:

(kmin[b,j] ≥ kmin[B,j]) et (kmax[b,j] ≤ kmax[B,j])

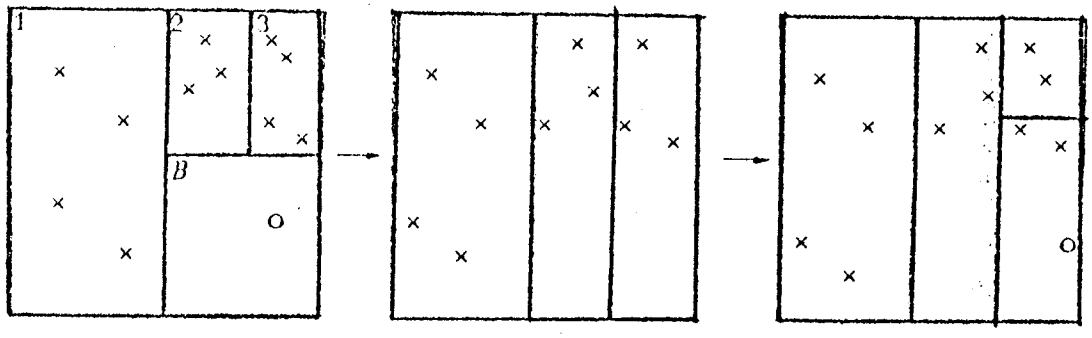
Pour chacun de ces blocs b, on fait:

kmax[b,k] ← kmax[B,k]

De plus, si B était le seul fils de son père, il faut indiquer que ce père n'a plus de fils: Coupure.Fils de Père de B ← 0.

B est le seul fils de son père si et seulement si on est dans les mêmes conditions que lors de la coupure, donc si leurs résumés sont identiques excepté que kmax[Père,k] = kmin[B,k].
Cette condition est aussi vérifiable par un filtre.

Exemple:



Les enregistrements restants de B doivent finalement être réinsérés dans le fichier. Comme ils seront forcément réinsérés dans l'ensemble de blocs que l'on vient de déterminer, il suffit

de travailler sur les résumés de cet ensemble, et pas sur les résumés de la totalité des blocs. Si l'on suppose une certaine continuité dans la répartition des valeurs des enregistrements, cet ensemble de blocs sera très petit (c'est l'ensemble des blocs ayant leur face k commune avec le seul bloc B). Ce travail d'insertion semble donc devoir être peu important.

Notons aussi que, pour diminuer le temps de balayage séquentiel du résumé, on peut construire un RESUME du RESUME, et ainsi de suite, de même que l'on crée plusieurs niveaux d'index dans les fichiers indexés classiques

6 2 3 3 Une généralisation des critères de recherche applicables aux résumés

Soit un fichier à n critères, organisé selon la méthode des résumés et une fonction f(x1, ,xn).

On recherche tous les enregistrements (x1, ,xn) du fichier tels que f(x1, ,xn) = 0.

Peut-on déduire de f une fonction g, telle que, pour tout bloc B du fichier, une condition nécessaire pour que B contienne un enregistrement désiré s'exprime sous la forme: g(RESUME (B)) ?

Le problème est soluble en particulier si la fonction f est telle que toutes ses dérivées partielles par rapport à chacune de ses variables sont de signe constant, autrement dit que f soit une fonction monotone de chacune de ses variables prises séparément.

Alors, la condition nécessaire recherchée est:

(f(y1, ,yn) $\leq$ 0) et (f(z1, ,zn) $\geq$ 0)

avec, pour tout i:

yi = kmin[B,i] si f est une fonction croissante de xi
yi = kmax[B,i] si f est une fonction décroissante de xi

et de même:

zi = kmax[B,i] si f est une fonction croissante de xi
zi = kmin[B,i] si f est une fonction décroissante de xi

Avec les mêmes définitions, la condition f(x1, ,xn) < 0 sur un enregistrement implique la condition f(y1, ,yn) < 0 sur le résumé d'un bloc, et f(x1, ,xn) > 0 implique f(z1, ,zn) > 0.
A la condition f(x1, ,xn) ≠ 0, correspond (f(y1, ,yn) > 0) ou (f(z1, ,zn) < 0).

Parmi les fonction f satisfaisant à la condition de monotonie, on peut citer les combinaisons linéaires des xi.
f(x1, ,xn) ↔ K + Somme(Ai × xi) où K et les Ai sont des constantes.

dont un cas particulier très utilisé est f(xi) ↔ xi + K.
De plus, étant donné un ensemble de fonctions f1, ,fp répondant à la condition de monotonie, la condition nécessaire associée à toute expression de 'ET' et de 'OU' sur ces fonctions: F(f1, ,fn) est tout simplement F(g1, ,gn) où les gi sont les fonctions déduites des fi comme on vient de l'expliquer.

Finalement, étant donné des critères de recherche très complexes de la forme:

((50 < age < 60) et (0,7 < (6×age + 10×ancienneté - 45×diplome) < 15,3) ou (salaire + 3×prime < 5000)

il est possible d'éliminer à priori tout bloc dont le résumé ne satisfait pas un critère déduit de celui-ci.

Il faut noter qu'une telle élimination n'est pas possible par l'exploitation de fichiers inverses classiques.

Finalement, l'organisation des fichiers par la technique des RESUMES apparait comme un outil bien adapté non seulement à l'interrogation des fichiers au sens classique, (opération de sélection de l'algèbre relationnelle), mais aussi à une certaine forme d'ANALYSE DES DONNEES.

6 3 UTILISATION DES FS POUR LES OPERATIONS DE JONCTION DES BASES DE DONNEES RELATIONNELLES

Etant donnés deux fichiers:
F1 : (Nom,Usine)
F2 : (Usine, Ville)
une JONCTION est une opération de recherche impliquant l'accès à un des fichiers à partir de clés provenant de l'autre. Exemple:

Donner les noms des personnes travaillant dans une usine située à Lille ou Paris

Ceci est une opération BINAIRE, et toutes celles étudiées dans les paragraphes précédents étaient des opérations UNAIRES. Comme le matériel dont nous disposons n'est pas fait pour s'occuper de DEUX fichiers à la fois, il va falloir réaliser cette opération en deux opérations unaires, de la meme manière que , dans une machine à accumulateur, A+B est réalisé par LOAD A, ADD B. Ici, le role d'accumulateur peut être joué par les mémoires à accès aléatoire disponibles: la mémoire du processeur et celle du filtre. Nous utiliserons celle du filtre. Exemple:

La première opération est un accès à F2: 'trouver les usines situées à LILLE ou PARIS'
Soient Us1, Us2, ..., Usn ces villes. La seconde question est alors:
Trouver les personnes travaillant dans les usines Us1 ou Us2 ou ... ou Usn.

A ce point, il faut distinguer deux cas:

-a) l'ensemble des Usi peut tenir dans la mémoire du filtre. Alors le 'ou' sur ces clès est compilé sous forme d'automate, et F1 est accédé avec cette question.

-b) l'ensemble des Usi est trop grand pour tenir dans le filtre, alors il faut le découper en morceaux et poser des questions successives sur F1.

Supposons que la première question ait sélectionné 1000 Usines, et que la mémoire du filtre très petite: 256 mots. Si une désignation d'usine tient sur 5 caractères en moyenne, on peut en mettre au moins 40 à la fois dans une question, et on doit donc faire 25 accès à F1. Avec des fichiers inverses, il faudrait faire 1000 (mille) accès au fichier F1.

Les deux solutions nécessitent évidemment le même nombre minimum d'accès aux blocs de données de F1 contenant les employés recherchés, mais les 25 accès des FSDS sont des accès au résumé de F1, et les mille accès du logiciel sont des doubles accès à un fichier inverse de F1.
Si F1 fait 50 millions de caractères, divisés en 3000 blocs environ d'une piste (~ 16 K), si un résumé tient sur 30 octets, alors le résumé de F1 tient sur 90000 octets, et un accès au résumé dure:

30 ms (temps d'accès) + 90 ms (temps de parcours)
et ceci 25 fois, soit 3 secondes.

Le logiciel a besoin de 1000 × 2 × 30 ms = 60 secondes.

Une solution logicielle qui simulerait la technique des résumés serait certes meilleure mais toujours beaucoup plus lente que les FS. Elle nécessiterait de plus une REFONTE TOTALE des SGBD actuels, ce qui serait un très mauvais choix stratégique: il est évident que les nouveaux SGBD devront être soit:

- la reprise des actuels améliorés localement et de manière interne par l'usage de matériel spécifique style FS

- de nouveaux systèmes conçus pour exploiter ces nouveaux matériels et bénéficiant non seulement de leur performance, mais aussi de la simplification logicielle qu'ils apportent.

Même avec une petite mémoire de filtre, cette technique est très intéressante pour les bases de données relationnelles. Pour l'améliorer encore, il faut:

- augmenter la taille de la mémoire du filtre, ou/et minimiser la mémoire nécessaire à la représentation d'un automate

- minimiser le temps de transformation de la réponse à la première question en une nouvelle question

Précisément, nous avons proposé la technique des automates tabulés (AT), qui ont les propriétés suivantes:

- leur taille peut ne pas être supérieure à deux fois celle qu'ils représentent (6 fois dans des cas extrêmes)

- ils peuvent être construits AU VOL, à la vitesse de débit d'un disque avec les technologies actuelles

En conséquence, le temps d'une opération de jonction entre deux fichiers peut se limiter au temps de lecture de ces deux fichiers, si la mémoire est assez grande pour contenir l'automate des valeurs de jonction.

Il est facile de voir que les mêmes résultats peuvent s'étendre aux opérations d'UNION et D'INTERSECTION de deux relations.

De plus, la mise sous forme d'automate d'un ensemble de valeurs résoud par construction le problème d'élimination des dupliqués après une PROJECTION.

En conclusion, le filtrage par automates semble être une

solution idéale pour exécuter TOUTES les opérations de base de l'algèbre relationnelle:

- la sélection
- la jonction sur égalité
- la projection
- l'union
- l'intersection

Toutes ces opérations peuvent s'effectuer AU VOL, donc dans le seul temps nécessaire à la lecture de leurs opérandes.

De plus, nous avons montré que , si les relations ne sont pas organisées comme des fichiers séquentiels mais comme des fichiers indexés, inversés ou selon le technique dite ici des ''résumés'', les filtres séquentiels constituaient aussi un outil très pratique pour explorer et mettre à jour de tels fichiers et leurs structures d'accès.

CHAPITRE 7

TOLERANCE AUX FAUTES D'ORTHOGRAPHE

7 TOLERANCE AUX FAUTES D'ORTHOGRAPHE

POSITION DU PROBLEME

En informatique classique, on utilise des mots de la langue naturelle essentiellement en deux occasions

- pour nommer des objets d'un système ou d'un langage (noms de fichiers, noms de variables, étiquettes d'instructions)

- pour nommer des données à l'intérieur d'un fichier (clés d'accès)

Dans les deux cas, toute faute d'orthographe est fatale, puisqu'il y a - à un instant donné et par définition - une bijection entre ces noms et les entités qu'ils désignent. Toute faute d'orthographe se traduit par une erreur à un stade ou à un autre de la vie de l'application.

Par contre, dans des domaines tels que l'informatique documentaire, ou plus généralement le traitement des données textuelles d'origine humaine, on ne peut pas s'accomoder de règles aussi strictes. Il faut tenter de tolérer dans l'ordinateur les types d'erreurs que l'homme tolère très bien, et en particulier les fautes d'orthographe. Traiter des données en texte naturel sans accepter de telles fautes créerait une situation aussi absurde que celle que nous connaitrions si la présence d'une seule faute d'orthographe dans un texte nous le rendait incompréhensible!

Bien sur, même dans les textes d'origine humaine, les fautes d'orthographe peuvent entrainer des ambiguités intolérables: c'est par exemple le cas des noms propres de personnes ou de lieux.

Néanmoins, même dans ce cas, il ne faut pas oublier qu'en informatique documentaire le traitement n'est que partiellement automatisé et que la machine ne fait qu'aider l'usager dans sa recherche. Ce n'est donc pas grave de lui donner une information inutile à la suite d'une faute d'orthographe, puisqu'il pourra finalement l'éliminer de visu s'il le désire. Cela vaut mieux que de lui interdire l'accès au document à cause d'une simple erreur de frappe. Par exemple, si on cherche les articles écrits par

DIJKSTRA , même si l'on n'a pas une idée très exacte de l'orthographe de ce nom, on est néanmoins humainement capable d'être attiré a la lecture d'un nom ressemblant a celui-ci. C'est une capacité analogue qu'il faut donner à la machine. On retrouve une notion fondamentale de RESSEMBLANCE, de DISTANCE entre des objets, (ici des mots), notion assez mal traitée par les systèmes d'information actuels.

Pour simplifier, nous nous limitons au problème suivant: si dans une question figure une expression telle que:

' NOM est MARTIN ou BERNARD ou MOREAU'

on aimerait que cette expression prenne la valeur vrai si le NOM est MARTINE ou MORO ou BERNHARD etc ...

Nous n'entrerons pas dans le domaine des linguistes ou des grammairiens qui utilisent des notions de ressemblance entre les mots faisant intervenir par exemple l'éthymologie, le contexte sémantique ou les régles et exceptions de conjugaison. [55]

Ici nous ne considérons que les FAUTES au sens propre du terme, et qui peuvent avoir deux sources:

- une cause mécanique (faute de frappe par exemple)
- une défaut de connaissance du langage

Notons qu'une faute peut exister aussi bien dans la question que dans les données interrogées, ou dans les deux à la fois.

En théorie, le problème se résoud très simplement: étant donné un mot recherché dans un enregistrement il suffit de donner la liste de TOUTES les orthographes que l'on accepte pour ce mot. Exemple:

NOM est MARTIN entraine en fait l'expression:
NOM est MARTIN ou AARTIN ou BARTIN ou MARDIN ... etc

Il suffit donc comme toujours de beaucoup de place en mémoire ...

Notons que cette méthode naive est néanmoins souvent employée en pratique pour des noms propres de lieux, avec la liste des fautes d'orthographe les plus fréquentes. Mais dans ce cas on suppose que l'on fixe a l'avance et les champs concernés et les orthographes possibles de ces champs.

En général, une telle énumération est trop couteuse en

mèmoire et en temps de compilation (meme si son temps d'exploration est acceptable grace a la dichotomie). Il faut donc un moyen de tolèrer la faute qui soit plus interprètè que compilè.

Nous passons en revue quelques solutions.

7 1 LA SOLUTION SARI [56]

C'est la plus classique. Elle consiste en deux choses:

1)- il existe un caractère spècial dit 'JOKER' qui peut remplacer n'importe quel autre. Lorsque l'on n'est pas certain d'une lettre, on la remplace par le JOKER:

MAR[]IN signifiera aussi bien MARLIN que MARTIN

2)- on retient non seulement les mots ègaux à celui donnè, mais aussi tous ceux qui commencent par ce mot, ce qui est une façon de tolèrer des fautes dans la fin du mot.

Ces deux techniques sont assez rustiques.
La principale critique à faire au joker est que, quand on fait une faute d'orthographe, on n'est pas censè connaitre l'endroit où l'on va la faire ...
D'autre part, dans le contexte du renseignement tèlèphonique qui est celui de SARI, le temps de rèflexion passè par les opèratrices à choisir la place des jokers dans les mots risque d'être pènalisant.

N.B. Les deux dispositifs de SARI sont très faciles a rèaliser avec un automate d'ètats finis.

7 2 L'ALGORITHME DE TUSERA HYAFIL [40]

Il permet d'accepter les fautes suivantes dans un mot:

-une premiére faute consistant soit de:

* une lettre fausse

* une lettre en trop
* une lettre en moins

-une seconde faute qui ne peut être qu'une lettre fausse

D'autre part, la première lettre du mot doit etre juste et les fautes ne peuvent se trouver que dans les parties linéaires des automates, non communes à plusieurs mots.

Le principe est en effet d'avancer dans l'automate même si la comparaison est fausse, pourvu que le nombre de fautes autorisé (1 ou 2) n'ait pas été atteint. Dans ces conditions, on ne peut évidemment CHOISIR entre plusieurs voies en cas d' échec, puisque le principe est de continuer 'tout droit'. Il faut donc que l'aiguillage vers un mot donné ait été fait avec les lettres de début du mot.

7 2 1 Principe et réalisation de l'algorithme de Tusera Hyafil

Le grand avantage de cet algorithme est qu'il ne coute rien en taille de l'automate, il suffit de préciser dans l'instruction du filtre associée à chaque lettre quel est le nombre de fautes que l'on tolére pour continuer quand même en cas de non égalité avec le caractére en entrée.

Le principe est le suivant:

Les instructions sont de la forme:

VC	A=	A≠	NF

où VC est le caractére recherché, NF le nombre maximum de fautes toléré (1 ou 2), A= est l'adresse suivante si ce nombre n'est pas dépassé aprés la comparaison, A≠ l'adresse suivante dans le cas contraire.

VC ne va pas être comparé seulement à CE comme d'habitude, mais aux TROIS derniers caractéres lus en entrée.

Par convention, on les appellera CE- , CE et CE+. CE- est le plus ancien et CE+ le plus récent. CE est considéré comme le ''vrai'' caractére courant.

A ces trois caractères, on associe trois compteurs qui vont compter le nombre d'échecs entre VC et le caractère auxquels ils sont associés. Soient K−, K et K+ ces compteurs. Leur incrémentation respecte les règles suivantes:

- ils sont remis à zéro à la fin du mot

- tant que VC = CE, on n'incrémente jamais K+ et K−

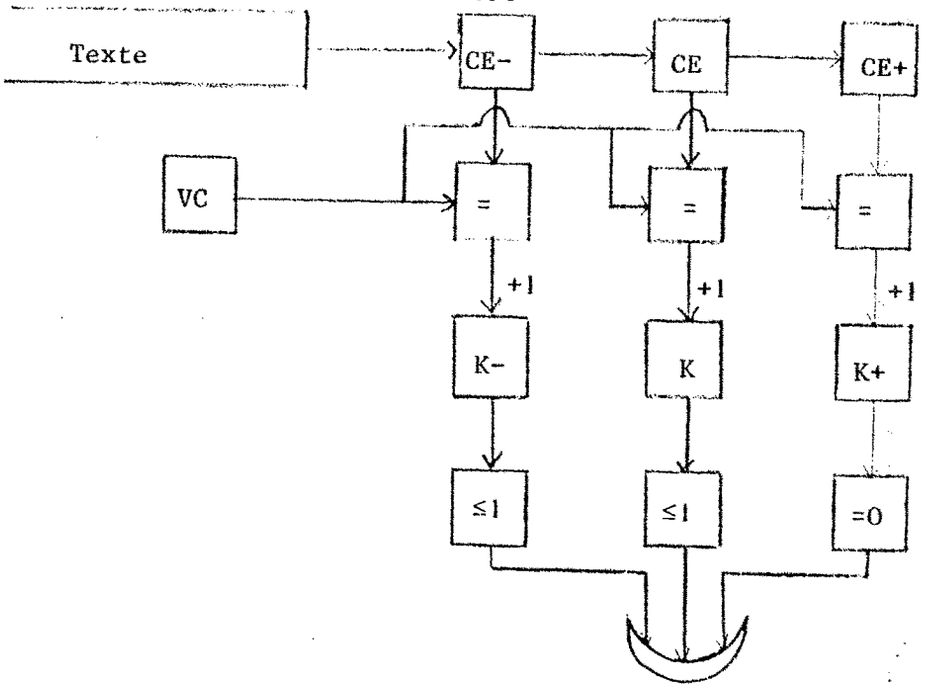
- dès que VC devient différent de CE, on autorise l'incrémentation de K+ et K−, chaque fois que VC ≠ CE+ ou VC ≠ CE− respectivement.

- à la fin du mot:

  * si tous les compteurs sont nuls, il n'y pas eu de faute dans le mot

  * si (K+ ≤ 1) ou (K− = 0) ou (K ≤ 1), il y a au plus une faute

  * si (K+ ≤ 2) ou (K− = 1) ou (K ≤ 2), il y a au plus deux fautes



1 faute au plus

N.B. La logique de bloquage de K+ et K- n'est pas dessinée.

7 2 2 Avantages et inconvènients de cet algorithme

Cette technique ne ralentit pas le processus de filtrage puisque les trois comparaisons se font en parallèle.

Elle ne nècessite pas de traitement spècial pour compiler l'automate de reconnaissance, il suffit de prèciser le nombre de fautes que l'on dèsire tolèrer dans chaque instruction.

Ses inconvènients sont:

- la rapidité due aux trois comparateurs n'est pas absolument nècessaire, car le nombre d'instructions à exècuter entre deux caractères est par dèfinition ègal à 1, puisque l'on doit être dans la partie linèaire de l'automate. Il y a une certaine redondance de matèriel.

- un manque de gènèralitè dans les fautes acceptèes (le type, le nombre et l'ordre des fautes ne sont pas quelconques)

- la nècessitè de disposer EN PLUS d'un signal 'fin de mot' . Dans la pratique, ceci implique qu'un autre mècanisme de

comparaison analyse le texte pour détecter le ou les délimiteurs de fin de mot; un tel mécanisme est typiquement un automate lui-même, qui peut avoir besoin de faire plusieurs comparaisons si il y a plusieurs délimiteurs possibles.

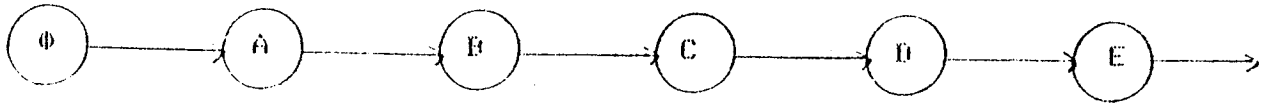
7 3 Proposition d'une solution utilisant les automates

Puisque notre but est de définir un opérateur de traitement de données 'crédible', donc peu couteux et aussi universel que possible, nous proposons une technique pour traiter les fautes d'orthographe qui tire le maximum de profit des automates, et garde deux de leurs qualités: vitesse et souplesse de programmation.

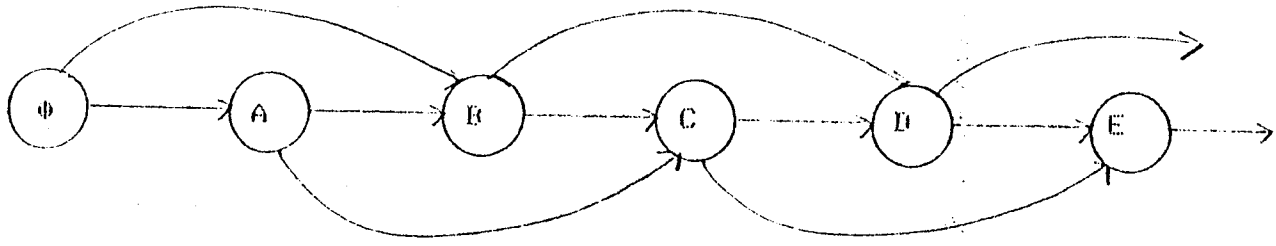
Le principe et le suivant:

Soit le mot 'ABCDE

L'automate qui le reconnait est:



L'automate qui accepte que certaines lettres manquent est:



N.B. Il n'accepte pas que deux lettres consécutives manquent.

Il faut compléter cet automate pour rechercher le(s) délimiteur(s). A chaque noeud il conviendrait donc d'ajouter une flèche vers la reconnaissance des délimiteurs. Pour plus de

clarté, ces flèches ne sont pas représentées, mais leur role est fondamental.

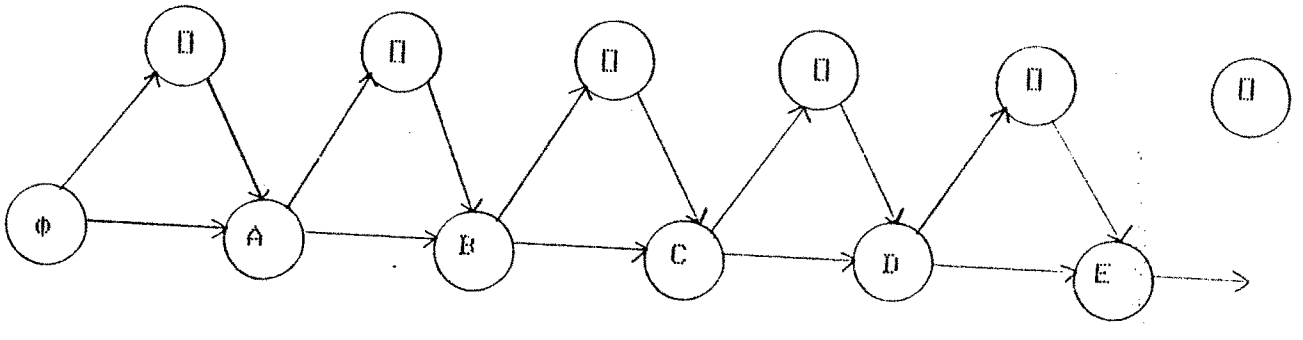
Lorsque l'on a trouvé la fin du mot, il faut un moyen de savoir combien de lettres on a sauté , ou inversement combien de lettres ont été correctement reconnues. Il suffit par exemple d'ajouter 1 à un compteur quand on prend le chemin normal. Et à la fin , ce compteur mesure la ressemblance entre le mot lu et le mot recherché.

On peut ensuite utiliser cette valeur de multiples manières pour la suite de l'évaluation de l'ensemble de la question.

Par exemple, au lieu de compter le nombre de mots reconnus, on peut cumuler des NOTES attribuées à chaque mot:

- +10 si zéro fautes
- +5 si 1 faute
- +2 si 2 fautes
- 0 si plus de 2 fautes ....

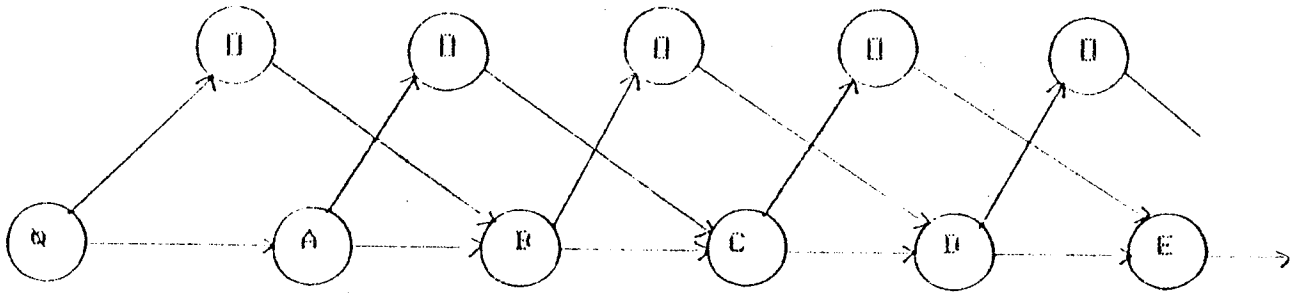
Voici l'automate qui accepte que figure une lettre en trop entre deux bonnes lettres:



N.B. □ dénote tous les caractères autres que ceux figurant sur les autres flèches partant du même noeud.

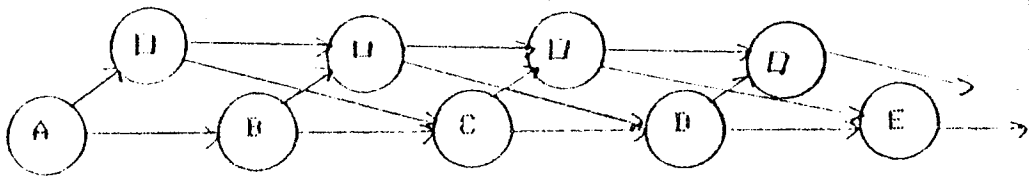
On peut par exemple ajouter 1 a un compteur chaque fois que l'on prend le chemin du haut, correspondant a une lettre en trop, et a la fin du mot, plus ce total est grand, moins le mot est ressemblant.

Voici l'automate qui accepte que des lettres soient fausses:

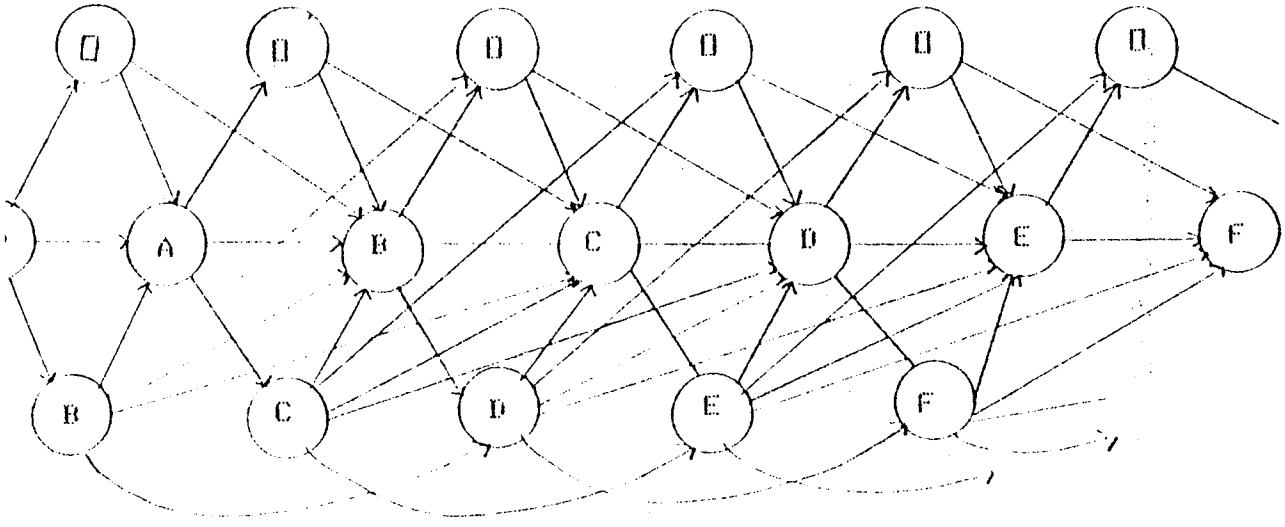


Là aussi, on peut compter le nombre de lettres fausses.

Pour accepter plusieurs lettres fausses consécutivement, il faut l'automate suivant:



Finalement, on peut fusionner ces trois automates en un seul:



Maintenant, il suffit d'affecter un 'bonus' ou un 'malus' a chaque branche de l'automate pour obtenir la mesure de ressemblance désirée.

Si l'on autorise deux 'faux pas' en dehors du chemin direct, on accepte ainsi les orthographes suivantes:

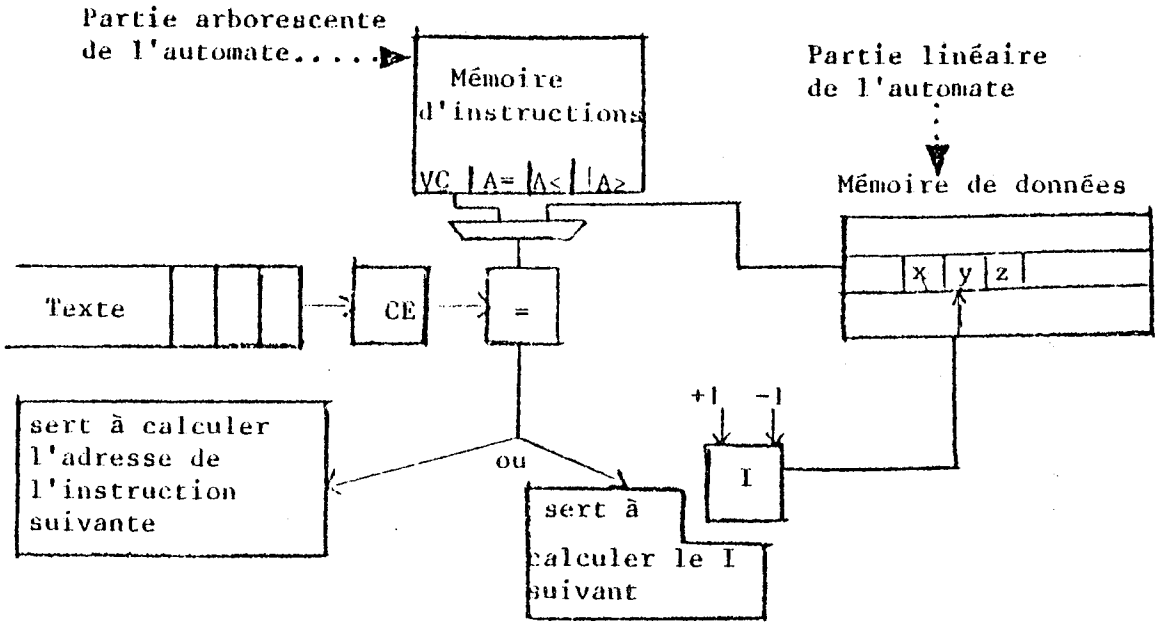
ACXD BABC XABC XBCD AXBXC AXBXD ... etc

Les restrictions le l'algorithme précédent sont levées, ainsi que les problémes de décalage du texte en entrée.

On voit que pour chaque lettre, on doit générer 7 instructions dans l'automate. Le compteur peut soit être intégré au filtre, soit être réalisé par un MODULE DE CALCUL avec alors des possibilités multiples pour calculer les bonus/malus.

En fait, il y a un moyen simple de supprimer ce facteur de 7, tout en gardant une solution logicielle, c'est de pouvoir appeler un sous-programme auquel on passe comme paramétres les valeurs des trois lettres suivant le point ou l'on est, et qui en résultat, choisira le lettre suivante (donc l'état suivant dans l'automate).

Il suffit de disposer d'un pointeur I sur la mémoire contenant la suite (LINEAIRE par hypothése) des caractéres du mot, et de comparer CE au caractére désigné par ce pointeur. Suivant le résultat de cette comparaison, on avancera ou reculera ce pointeur, et on calculera les bonus/malus éventuels.



Les différences avec l'algorithme précédent sont les suivantes:

- au lieu de comparer UN caractère de l'automate avec PLUSIEURS caractères en entrée, on compare LE caractère en entrée avec PLUSIEURS caractères de l'automate. Les problèmes de décalage sont supprimés, car la mémoire de l'automate a un accès aléatoire, alors que celle des caractères en entrée a un accès strictement série.

- les comparaisons multiples ne sont pas faites en parallèle mais en série, et sous le controle de l'algorithme PROGRAMME et non cablé qui va avancer ou reculer le pointeur sur la mémoire de l'automate.

Il y a alors toutes facilités pour étendre d'une manière purement logicielle:

- la gravité de chaque faute individuelle: nombre de lettres fausses consécutives admis

- le nombre total de fautes différentes admis

- tout en ayant de plus grandes possibilitée et un plus faible cout en matériel, cet algorithme a des performances suffisantes en pratique, comme le montrent les résultats de simulation exposés plus loin.

7 4 Mise en oeuvre et évaluation de cet algorithme.

Parmi les différentes réalisations possibles de cet algorithme, nous en présentons une adaptée à l'architecture du type ''filtre intégré au module de calcul''.

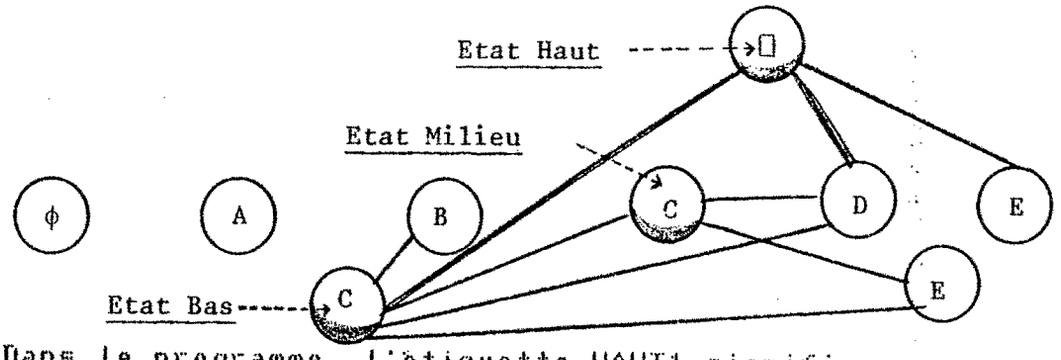
Dans une telle architecture, les parties arborescentes de l'automate sont représentées par des INSTRUCTIONS et les parties linéaires (celles qui nous intéressent ici) sont représentées par des DONNEES, adressables entre autres par un registre spécial que nous appelons I.

Voici une programmation possible de l'algorithme de tolérance des fautes.

- il n'utilise pas de compteurs de fautes, mais cette notion est COMPILEE: au début, on part avec un crédit de n fautes. Lorsque l'on découvre une faute, le programme se branche vers un endroit où le crédit n'est plus que de n-1, et ainsi de suite jusqu'à ce que le crédit soit épuisé, ou jusqu'à ce que la fin de mot soit atteinte dans le texte en entrée ou dans l'automate. L'exemple est donné avec un crédit de départ de 2 fautes.

A un instant donné, l'état de l'algorithme est soit 'haut', 'milieu' ou 'bas', par allusion au dessin de l'automate:

Lorsque I pointe sur ''C'', les états sont ainsi définis:



Dans le programme, l'étiquette HAUT1 signifiera par exemple: On est dans l'état haut, et le crédit est de 1 faute.

Voici le programme de principe, qu'il suffit de générer N+1 fois, en faisant varier i de 0 à N si on accepte au maximun N

fautes.

Pour toutes les étiquettes indicées de la forme Haut(i), si i vaut -1, le crédit de fautes est épuisé, la comparaison est fausse.

L'étiquette Finmot est atteinte lorsque le mot testé est égal à la totalité ou au début du mot cherché, aux fautes d'orthographes près.

```
Haut(i) :   Lire CE suivant; si c'est un séparateur, aller à
Finmot
            I←I+1; si MCI] est un séparateur, aller à
Succès(i)
            Si CE = MCI], aller à Milieu(i)
            I←I+1; si MCI] est un séparateur, aller à
Succès(i)
            Si CE = MCI], aller à Milieu(i)
            I←I-1; aller à Haut(i)

Milieu(i):  Lire CE suivant; si c'est un séparateur, aller à
Finmot
            I←I+1; Si MCI] est un séparateur, aller à
Succès(i)
            Si CE = MCI], aller à Milieu(i)
            I←I+1; Si MCI] est un séparateur, aller à
Succès(i-1)
            Si CE =MCI], aller à Bas(i-1)
            I←I-2; aller à Haut(i-1)

Bas(i):     Lire CE suivant; si c'est un séparateur, aller à
Finmot
            I←I-1
            Si CE = MCI] aller à Milieu(i)
            I←1+1
            Si CE = MCI]  aller à Milieu(i)
            I←I+1; si MCI] est un séparateur, aller à
Succès(i)
            Si CE =  MCI] aller à Milieu(i)
            I←I+1; si MCI] est un séparateur, aller à
Succès(i-1)
            Si CE = MCI] aller à Bas(i-1)
            I←I-2; Aller à Haut(i-1)
```

Pour évaluer les performances de cet algorithme, nous avons recherché le mot ''TIERRY'' dans la liste des prénoms du calendrier (considérée comme un texte à analyser et non pas comme une liste de mots-clés).

Le résultat est que, pour chaque caractère en entrée, il a fallu faire:

- 1 comparaison pour tester si il était égal à un séparateur (ici supposé unique).

- 0,867 comparaisons en moyenne avec un caractère de l'automate.

Ce chiffre extrêmement faible s'explique simplement par le fait que quasimment tous les mots ont une mauvaise orthographe, et que, en général:

- au bout d'une lettre ,on est dans l'état Haut, où seules deux comparaisons sont effectuées

- au bout de 3 ou 4 lettres, le crédit de fautes est épuisé, et l'on abandonne les comparaisons pour attendre la fin du mot ...

En particulier, il ne faut pas être effrayé par les quatre comparaisons successives de l'état bas, elles ne se produisent que très rarement.

Si de plus, il y a de nombreux séparateurs possibles, il faudra plusieurs comparaisons pour les rechercher, et les comparaisons avec les caractères de l'automate prennent un temps négligeable. Cette remarque plaide aussi pour l'utilisation de tables de transcodage donnant le type de chaque caractère, ou transcodant tous les séparateurs en un seul.

Finalement, on retrouve ici la même remarque que pour les performances des automates en général: il y a très peu de travail à faire en moyenne. Il est donc inutile de

surdimensionner le matériel. On peut rester très ''programmable'', cela coutera moins cher et permettra beaucoup plus de choses.

En conclusion, avec des données textuelles, le travail de reconnaissance d'un mot se décompose donc en moyenne en trois parties:

- une courte partie de recherche arborescente

- une courte partie de recherche linéaire avec fautes autorisées

- une longue partie d'attente de la fin du mot

CHAPITRE 8

UN COMPILATEUR DE QUESTIONS

POUR FILTRES SEQUENTIELS

8 UN COMPILATEUR DE QUESTIONS POUR FILTRES SEQUENTIELS

Ce chapitre discute de la définition d'un langage de requêtes pour une 'machine filtrante', et expose la solution retenue et sa réalisation.

8 1 DIFFICULTES DE SPECIFICATION ET DE REALISATION D ' UN COMPILATEUR GENERAL

Etant donnée une machine quelconque, il ne faut pas parler d'un compilateur permettant d'utiliser cette machine, mais d'un compilateur permettant d'utiliser un langage de programmation donné, langage qui, en général, n'utilisera qu'une partie des ressources de la machine. De plus, la sémantique du langage n'est pas forcément liée à celle du langage de la machine. Ces remarques restent vraies même si, comme dans notre cas, la machine n'est pas universelle, mais spécialisée. Avant d'écrire un compilateur, il faut définir le langage à compiler.

La spécification d'un tel langage n'est pas une chose facile; de nombreux problèmes se posent, parmi lesquels il faut citer:

1)- la nouveauté du langage envisagé: tous les langages d'interrogation existants travaillent sur des enregistrements ayant des structures plates ou hiérarchiques simples (n-uples des relations, entités de Socrate). Ici, nous voudrions profiter du fait que le format des enregistrements sera analysé par un automate à piles pour autoriser le maximum de libertés dans sa définition.

Les structures de données devenant ainsi plus riches et non figées au départ, la syntaxe et la sémantique des critères de sélection seront plus difficiles à définir.

2)- la nouveauté des contraintes dues à l'architecture de la machine:

* les caractères d'un enregistrement ne sont lus qu'une seule fois, dans l'ordre où ils se présentent. Sur une machine classique, l'enregistrement est rangé dans une mémoire aléatoire, donc peut être analysé par un algorithme quelconque.

* on doit (ou on souhaite) travailler ''au vol''. Le nombre d'opérations à faire en moyenne sur chaque caractère est borné. Il s'agit d'une contrainte tout à fait inhabituelle pour un langage et son compilateur: le code objet doit non seulement être correct sémantiquement, mais aussi temporellement.

3)- les particularités du filtre de la machine-objet utilisé,(celui de la machine TREFLE) [36].

* grandes facilités lexicales et syntaxiques

* faibles facilités sémantiques : limitées au calcul booléen, et sans la récursivité des structures de données (tables de bascules, tables de vérité) qui permettraient véritablement de profiter des facilités lexicales et syntaxiques.

4)- l'étendue des applications visées:

* applications documentaires textuelles (enregistrements peu formattés, utilisateurs non spécialistes)

* applications bases de données classiques (enregistrements très formattés, utilisateurs spécialistes)

* applications nouvelles: analyse de structures de données récursives (listes, arbres)

En conclusion, il s'agit de définir un langage et son compilateur pour des applications variées, respectant des contraintes de temps, pour une machine non universelle.

Dans un premier temps, nous nous sommes imposés un certain nombre de restrictions pour aboutir à un compilateur simplifié, mais au moins capable d'être utilisable dans des applications significatives. A la lumière de cette expérience, nous espérons voir plus clairement dans quel sens il conviendra de la généraliser.

8 2 UN COMPILATEUR SIMPLIFIE

8 1 Architecture globale du compilateur.

Un premier langage LS (Langage de Structure) permet de définir la structure des enregistrements, sous une forme BNF simplifiée, et imposant certaines déclarations explicites, comme celles de délimiteurs de début et de fin.

Pour chaque fichier, il faut donner la définition de ses enregistrements. Ce travail n'est pas supposé être à la charge de l'utilisateur final (celui qui pose les questions), mais à celle de l'''administrateur''' de la base de données, selon le terme consacré.

La description est faite une fois pour toutes à la création du fichier, et l'utilisateur final n'en connait pas les détails (ordre des différents champs, valeur des délimiteurs). Il ne connait que la structure logique:

- noms des champs
- relations d'inclusion entre ces champs

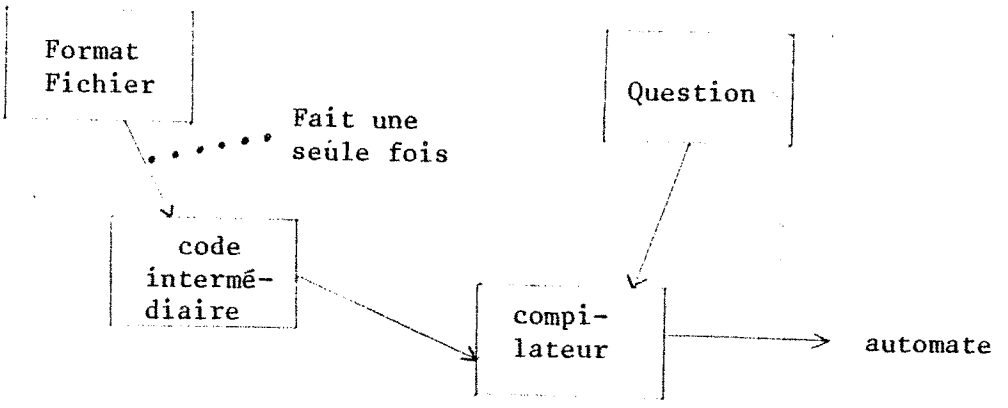
Le second langage est celui qui permet à l'utilisateur de formuler sa requête. On l'appelle LR (Langage de Requêtes)

Il dispose pour cela de facilités pour

- nommer les champs
- rechercher des sous-chaines dans un champ
- faire des comparaisons $< \leq = \geq > \neq$
- exprimer des conditions booléennes

Le code objet dépend à la fois du texte de la question et du format du fichier.

Pour minimiser le temps de compilation, le texte décrivant le format du fichier n'est pas réanalysé à chaque question, mais traduit au départ en un code intermédiaire et un ensemble de tables qui sont ensuite utilisées pour guider la compilation de la question.



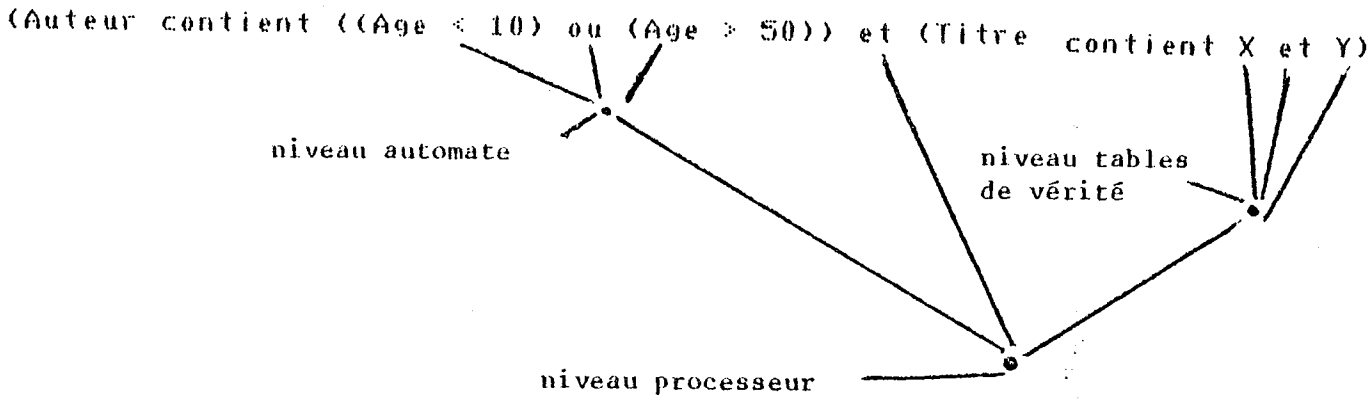
Le compilateur accepte trois niveaux de calcul booléen imbriqués;

- un sur les comparaisons $< \leq = \geq > \neq$, réalisé au moyen de transitions appropriées de l'automate.

- le mécanisme d'indexation de tables de vérité du filtre

- un niveau supplémentaire réalisé par le processeur de controle du filtre, qui reçoit les caractères sortant du filtre.

Le filtre ne calcule donc que des conditions booléennes partielles. Il envoie ces résultats au processeur de controle qui calcule la condition finale. Exemple;



En raison de son caractère expérimental, ce compilateur a été écrit en APL, ce qui présente des inconvénients et des avantages:

- inconvénient : son temps d'exécution est important, ce qui le rend en pratique trop lent (sur de petites machines) ou trop couteux (sur de grosses machines). Néanmoins, un effort a été fait pour utiliser un APL le plus simple possible (tableaux à une dimension au plus) afin de faciliter sa réécriture dans des langages comme PASCAL, ou sa traduction en langage machine par la méthode APL2M que nous avons développée [57] [58]. De plus, des travaux anciens et récents permettent d'espérer pouvoir utiliser des compilateurs APL à court terme [59] [60].

- avantages: l'utilisation de la technique APL2M consistant à réutiliser la syntaxe d'un langage existant (APL en l'occurence) pour en définir une nouvelle, apporte à la fois une diminution considérable du travail et une souplesse accrue pour la communication homme-machine:

Par exemple, quand nos écrirons:

TROUVER ARTICLES AVEC (AUTEUR CONTENANT 'A' OU 'B') ET (DATE EGALE 1950)

ce texte sera en fait une EXPRESSION APL, où TROUVER est une fonction monadique, AVEC, OU , ET, EGALE et CONTENANT des fonctions dyadiques, ARTICLE, AUTEUR et DATE sont des variables APL, et 'A' , 'B' et 1950 sont des constantes.

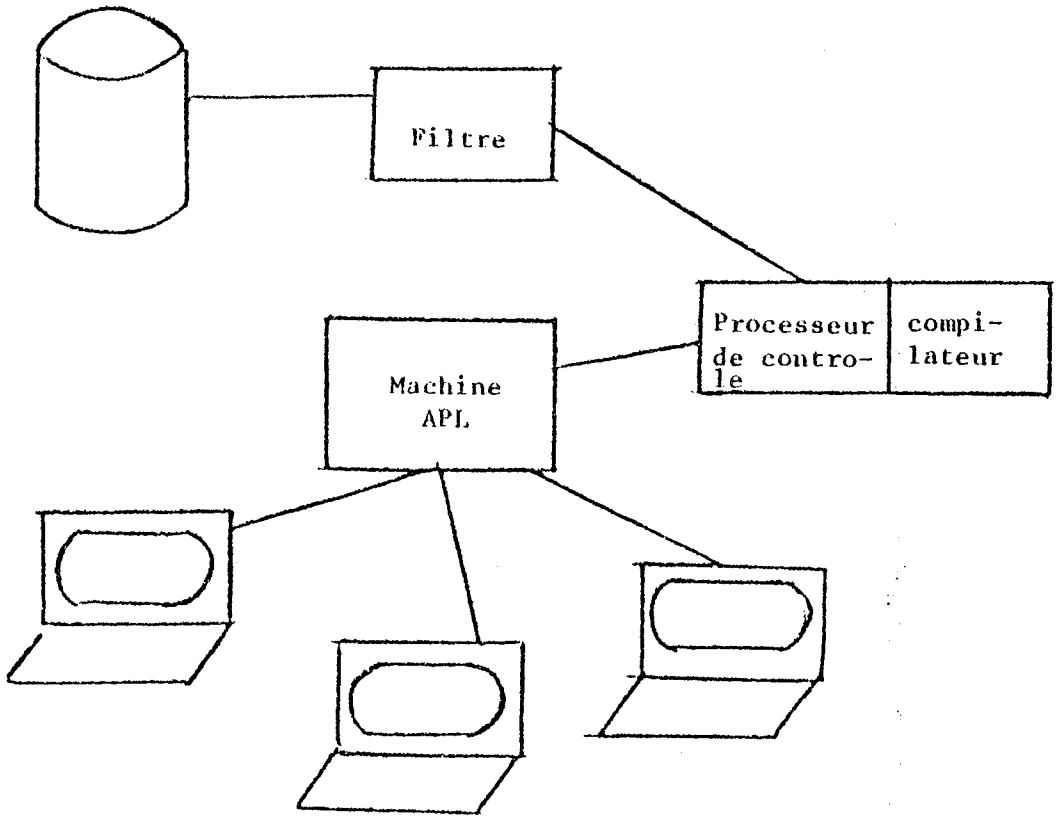
C'est l'EXECUTION de ce texte (et non son analyse par un compilateur classique) qui va provoquer la génération du code objet, donc sans qu'aucun travail d'analyse lexicale et syntaxique n'ait été fait.

Cette technique, qui s'apparente à une macrogénération commandée par APL, offre de multiples facilités quasi gratuite. Par exemple, la question ci-dessus pourrait aussi bien s'écrire:

```
LISTE← 'A' OU 'B'
COND1← AUTEUR CONTENANT LISTE
COND2← DATE EGALE 1950
TROUVER ARTICLE AVEC COND1 ET COND2
```

Ce qui est remarquable ici est que ces facilités d'écriture n'ont demandé aucun investissement de programmation.

Un autre avantage est que APL est plus qu'un langage, c'est un système conversationnel permettant de gérer l'ensemble des fonctions et des variables (comme notre variable LISTE ci-dessus) dans un 'espace de travail' qui est conservé d'une session à l'autre. Avec l'apparition de machines APL sur microprocesseurs ou sur miniordinateurs, on peut envisager que les utilisateurs accéderont à une machine base de données à travers une machine APL très ''conviviale'', quitte à déporter vers le processeur de controle l'exécution des parties les plus longues de la compilation:



8 3 LE LANGAGE DE STRUCTURES: LS

La structure d'un fichier est décrite par une grammaire à contexte libre.

Les noms des règles employés dans cette description pourront ensuite être utilisées dans les critères de filtrage exprimés dans le langage LR défini plus loin.

8 3 1 Déclaration des règles syntaxiques

Pour simplifier la compilation ainsi que le travail de description des enregistrements de manière syntaxique, nous introduisons une notion de SEPARATEURS.

Pour toute règle non terminale, il faut déclarer ses ensembles de séparateurs de début et de fin.

Un ensemble de sèparateurs est un ensemble de chaines de caractères. Exemple:

'ABCDIEFGITRFIUIO'

est un ensemble de 4 chaines, chacune sèparèe par un 'I'.
La syntaxe de dèclaration des sèparateurs d'une règle est la suivante:

< sèparateurs de dèbut> DEBUT < nom de règle > FIN < sèparateurs de fin >

Exemple:

'//ItI' DEBUT 'TITRE' FIN '\\I↓↓'

8 3 2 Dèfinition des règles de syntaxe

Cette dèfinition diffère de la forme normale de Backus en un certain nombre de points:

- le ''::='' est remplacè par ''EST''
- le ''I'' (alterative) est notè ''OU''
- la succession de deux èlements (implicite en BNF) est notèe par ''PUIS''
- les noms de règles non terminales ne sont pas entre ''<'' et ''>''
- la notation (PLUSIEURS α) signifie que l'expression α peut se rèpèter zèro ou plusieurs fois.

Par exemple, la description suivante:

A EST (B OU C OU D) PUIS (E OU F)
Y EST PLUSIEURS X

est èquivalente en BNF à:

< A > ::= (<B> I <C> I <D>) (<E> I <F>)

< Y > ::= (. |<X> (. |<Y>))

De plus, les délimiteurs, déclarés explicitement comme expliqué précédemment, jouent un rôle IMPLICITE lors de la définition des règles.

Toute DECLARATION de règle de la forme :

α DEBUT 'NOMREGLE' FIN ω

est équivalente à une DEFINITION qui serait de la forme:

NOMREGLE EST α PUIS NOMREGLE1 PUIS ω

où NOMREGLE1 désigne le ''corps'' de la règle, sans les délimiteurs de début et de fin.

Dans ces conditions, l'utilisation de NOMREGLE en partie gauche d'un ''EST'' est une convention d'écriture; c'est en fait NOMREGLE1 qui est alors défini. Finalement::

α DEBUT 'NOMREGLE' FIN ω
NOMREGLE EST ρ

est équivalent à:

NOMREGLE EST α PUIS ρ PUIS ω

Voici un exemple complet de description dans le langage LS:

```
'I' DEBUT 'ARTICLE' FIN 'A'
'J' DEBUT 'TITRE' FIN 'B'
'K' DEBUT 'AUTEUR' FIN 'C'
'L' DEBUT 'TEXTE' FIN 'D'
'M' DEBUT 'AGE' FIN 'E'
'N' DEBUT 'NOM' FIN 'F'
'O' DEBUT 'DATE' FIN 'G'
'P' DEBUT 'LIEU' FIN 'H'

  ARTICLE EST PLUSIEURS TITRE OU AUTEUR OU TEXTE
  TITRE EST PLUSIEURS DATE OU LIEU
  AUTEUR EST PLUSIEURS AGE OU NOM
```

8 4 LE LANGAGE DE REQUETES LR

Le langage de requête permet d'exprimer des conditions de sèlection d'enregistrements dont le format a été défini au moyen du langage LR.

Le langage LR permet de spècifier:

a)- des comparaisons $< \leq = \geq > \neq$ entre le contenu d'une règle et une constante

b)- l'appartenance d'une sous-chaine constante au contenu d'une règle

c)- des expressions booléennes portant sur la validité de conditions de type a) et b)

Examinons ces points en détail.

Voici la définition des critères descriptibles en LR.

< atome de comparaison > ::= (< | ≤ | = | ≥ | > | ≠) <constante >

< terme de comparaison > ::= TOUTE EXPRESSION BOOLEENNE DE < atome de comparaison >

Exemple:
'=64' OU ('≤56' ET '≥ 42')

< expression de comparaison > ::= <nom de règle> EST < terme de comparaison >

Exemple:

AGE EST '<30' OU '>40'

< expression de comparaison étendue > ::= <nom de règle > DONT expression booléenne de <expression de comparaison >

Exemple:

AUTEUR DONT (AGE EST '=56') et (SALAIRE EST '>1000' ET '<2000')

Ceci concernait les comparaisons entre une constante et des parties d'un enregistrement dont les valeurs sont munies d'une relation d'ordre. Voici maintenant des expressions concernant les TEXTES.

< expression textuelle > ::= < nom de règle > CONTENANT < expression booléenne de chaines de caractères >

Exemple:

TEXTE CONTENANT ('PARIS' et 'MARSEILLE') ou ('NICE' et

non 'LILLE')

Nous pouvons maintenant définir la forme générale d'une REQUETE du langage LR:

< requete > ::= TROUVER < nom de règle > AVEC expression booléenne de < expression textuelle > et de < expression de comparaison étendue >.

Exemple:

TROUVER ARTICLES AVEC (TEXTE CONTENANT ('AAA' OU 'BBB') ET 'CCC' OU 'DDD') ET AUTEUR DONT ((AGE EST '<50' OU '≥60') ET (NOM EST '≠SMITH'))

8 5 EXEMPLES DE CODES GENERES PAR LE COMPILATEUR

Le code objet utilise un sous-ensemble du jeu d'instructions du filtre d'une machine existante [36] dont le format est:

VC	A=	A<	A>	F1	N	TR

Voici la sémantique de cette instruction:

VC, A= , A< , A> ont leur sens habituel

Si F1 = 16, il faut lire le caractère CE suivant quand CE ≠ VC

Si F1 = 4, il faut mettre à 1 la bascule dont le numéro est donné par le champ N

Si F1 = 20, (16 + 4), il faut faire les deux actions

ci-dessus

Si TR = 2, il faut envoyer en sortie du filtre la valeur du champ N, modifiée partiellement par le résultat de l'indexation des tables de vérité booléennes.

Si VC = 255, la comparaison est considérée comme bonnne, quelle que soit la valeur de CE, et le CE suivant n'est pas lu, sauf si F1 ≥ 16

Un ensemble de 8 bascules et la table de 256 bits associée est attribué à chaque règle située en partie gauche d'une « expression de comparaison étendue» ou d'une « expression textuelle ».

A la fin d'une telle règle, il faut sortir le résultat correspondant à UN SEUL ensemble (bascules, tables). La machine objet sortant seulement l'encodage prioritaire de TOUS les résultats (c'est à dire le numéro du résultat à 1 le plus à gauche), il est nécessaire de procéder ainsi:

Dans chaque ensemble de 8 bascules, la bascule de poids faible est mise à 1 lors de la détection de la fin de la règle, avant l'envoi du résultat. Les tables de vérité sont compilées de façon à rendre toujours un zéro si cette bascule est à zéro, et à rendre le résultat désiré si elle est à 1. Ainsi, à la fin d'une règle, l'encodage prioritaire est différent de zéro seulement si le résultat de LA table désirée est 1.

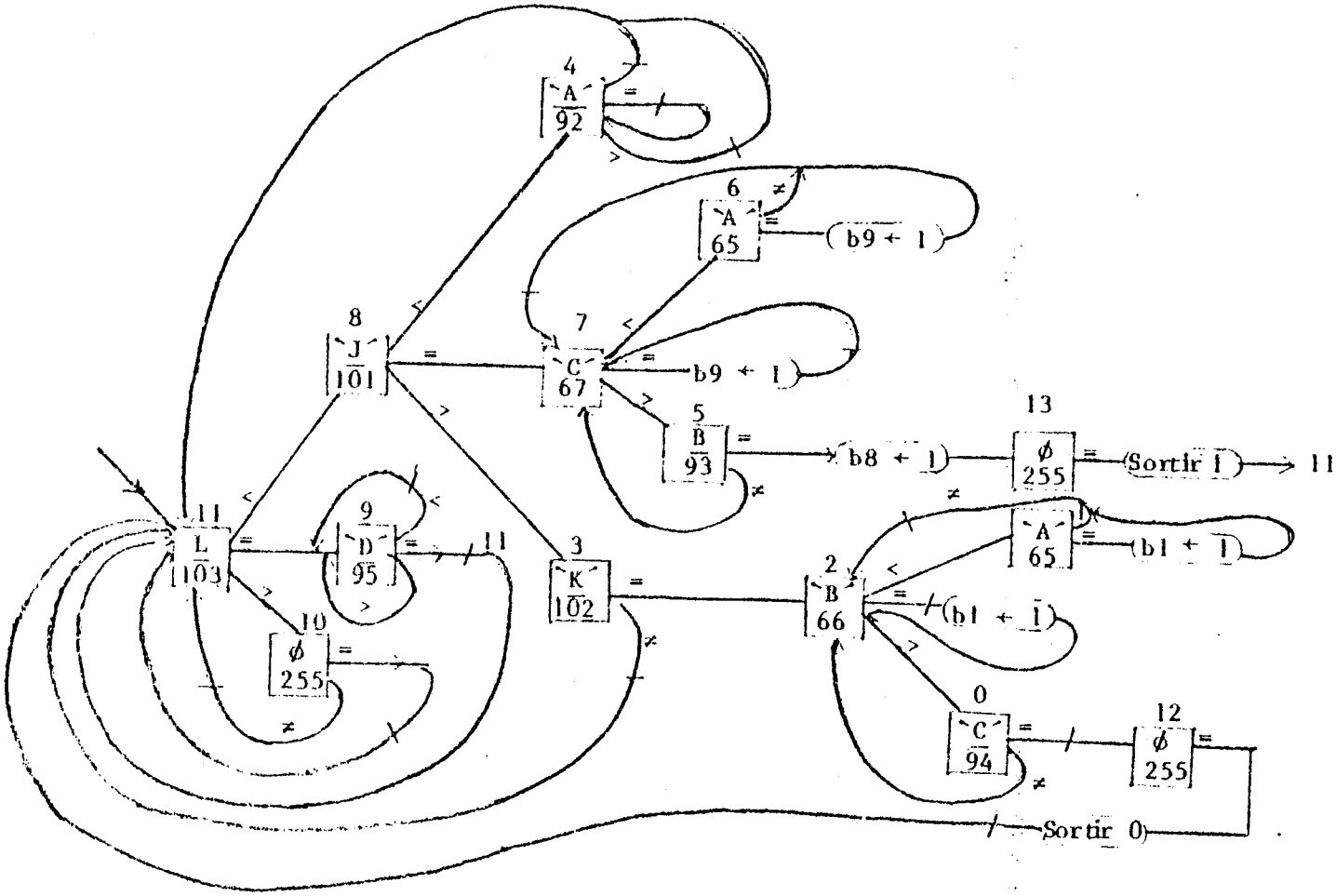
Voici deux exemples de questions avec le détail du code généré

Premier exemple:

TROUVER ARTICLE AVEC (AUTEUR CONTENANT 'A' OU 'B') OU (TITRE CONTENANT 'A' OU 'C')

N.B. Le format du fichier utilisé est celui donné en exemple dans le § sur le langage LS.

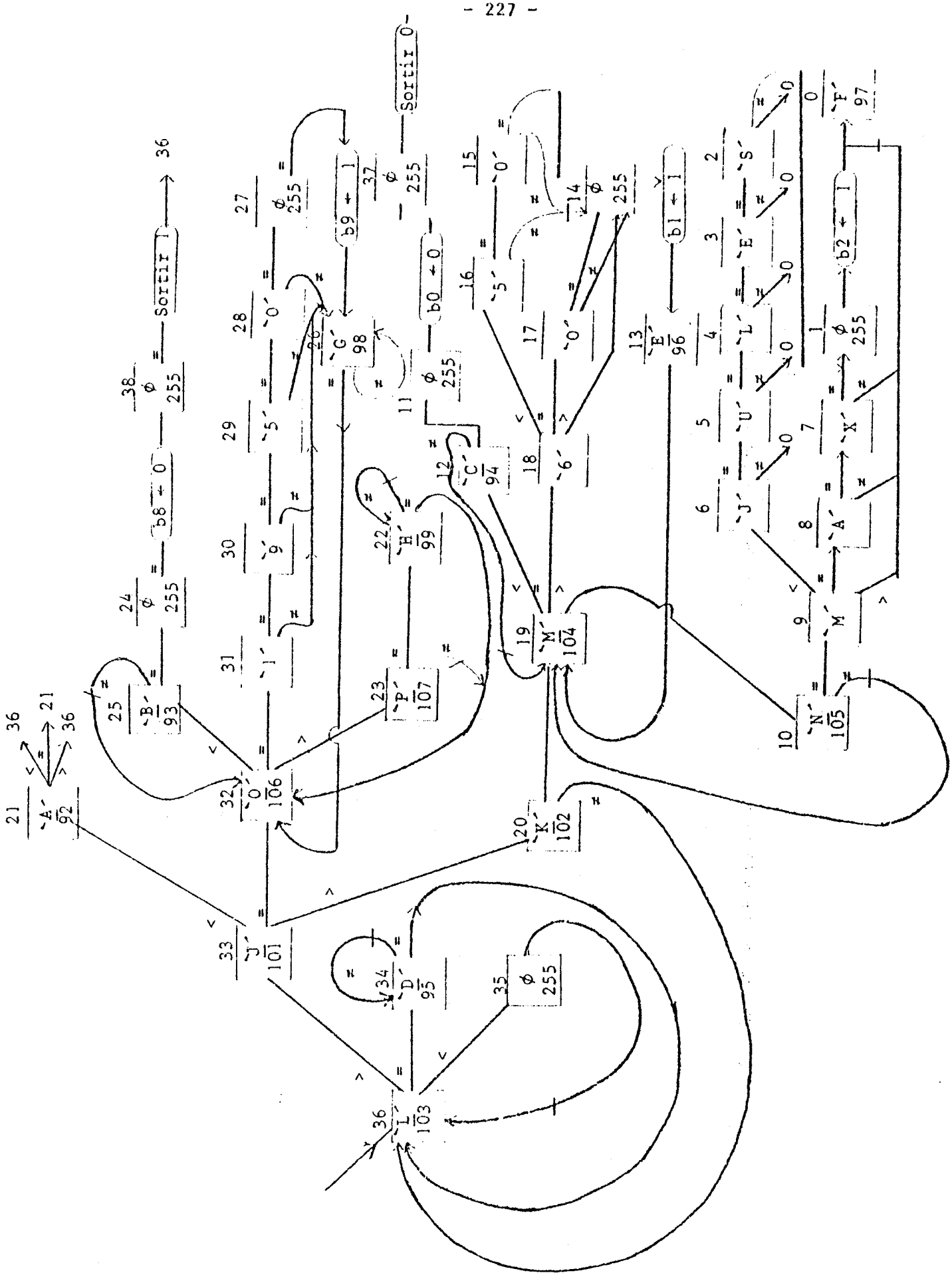
0	C	94	12	2	2	20	0	0
1	A	65	2	2	2	20	1	0
2	B	66	2	1	0	4	1	0
3	K	102	2	11	11	16	0	0
4	A	92	4	11	11	16	0	0
5	B	93	13	7	7	20	8	0
6	A	65	7	7	7	20	9	0
7	C	67	7	6	5	4	9	0
8	J	101	7	4	3	0	0	0
9	D	95	11	9	9	16	0	0
10	Φ	255	11	11	11	16	0	0
11	L	103	9	8	10	0	0	0
12	Φ	255	11	11	11	0	0	2
13	Φ	255	11	11	11	0	1	2


4
A
92
6
A
65
b9 ← 1
8
J
101
7
C
67
b9 ← 1
5
B
93
b8 ← 1
13
ϕ
255
Sortir 1
11
11
L
103
9
D
95
11
3
K
102
2
B
66
A
65
b1 ← 1
b1 ← 1
10
ϕ
255
0
C
94
12
ϕ
255
Sortir 0

Deuxième exemple:

TROUVER ARTICLE AVEC (AUTEUR DONT (AGE EST '>50' OU '<60') OU (NOM EST '=MAX' OU '=JULES')) OU (TITRE DONT DATE EST '=1950)

0	F	97	19	0	0	16	0	0
1	Φ	255	0	0	0	4	2	0
2	S	83	1	0	0	0	0	0
3	E	69	2	0	0	0	0	0
4	L	76	3	0	0	0	0	0
5	U	85	4	0	0	0	0	0
6	J	74	5	0	0	0	0	0
7	X	88	1	0	0	0	0	0
8	A	65	7	0	0	0	0	0
9	M	77	8	6	0	0	0	0
10	N	105	9	19	19	16	0	0
11	Φ	255	37	0	0	4	0	0
12	C	94	11	19	19	16	0	0
13	E	96	19	13	13	16	0	0
14	Φ	255	13	0	0	4	1	0
15	0	119	14	14	14	0	0	0
16	5	124	15	14	14	0	0	0
17	0	119	14	14	14	0	0	0
18	6	125	17	16	14	0	0	0
19	M	104	18	12	10	0	0	0
20	K	102	19	36	36	16	0	0
21	A	92	21	36	36	16	0	0
22	H	99	32	22	22	16	0	0
23	P	107	22	32	32	16	0	0
24	Φ	255	38	0	0	4	8	0
25	B	93	24	32	32	16	0	0
26	G	98	32	26	26	16	0	0
27	Φ	255	26	0	0	4	9	0
28	0	119	27	26	26	0	0	0
29	5	124	28	26	26	0	0	0
30	9	128	29	26	26	0	0	0
31	1	120	30	26	26	0	0	0
32	O	106	31	25	23	0	0	0
33	J	101	32	21	20	0	0	0
34	D	95	36	34	34	16	0	0
35	Φ	255	36	36	36	16	0	0
36	L	103	34	33	35	0	0	0
37	Φ	255	36	36	36	0	0	2
38	Φ	255	36	36	36	0	1	2


Sortir 0
Sortir 1
A 92
B 93
C 94
D 95
E 96
F 97
G 98
H 99
I 101
J 101
K 102
L 103
M 104
N 105
O 106
P 107
b0 ← 0
b1 ← 1
b2 ← 1
b8 ← 0
b9 ← 1
Φ 255

8 6 UN REGARD SUR D'AUTRES EXEMPLES D'UTILISATION DE L'ANALYSE SYNTAXIQUE COMME OUTIL DE PROGRAMMATION.

L'idée d'utiliser l'analyse syntaxique pour diriger l'exécution d'un programme n'est pas nouvelle. On la trouve de manière informelle dans les ''compilateurs de compilateurs'' et de manière plus formelle dans le langage SNOBOL [94].
Examinons ces deux exemples.

8 6 1 Utilisation des compilateurs de compilateurs comme outils de programmation généraux.

Ces produits sont essentiellement des langages permettant d'insérer des appels de sous-programmes d'un langage existant dans une description grammaticale (sous forme BNF en général) des données analysées. A partir de cette description ainsi enrichie d'ACTIONS SEMANTIQUES, le compilateur de compilateur construit un analyseur syntaxique qui appellera les sous-programmes indiqués lorsque la règle qui les précède dans la description aura été reconnue.

Ce style de programmation est souvent généralisé à des problèmes n'ayant rien à voir avec l'analyse syntaxique. Il permet en effet d'exprimer clairement les principales structures de controle:

- la séquence
- l'itération
- l'alternative
- l'imbrication

Par exemple, il n'est pas rare de le voir utilisé dans les compilateurs non seulement pour la phase d'analyse syntaxique -pour laquelle il est explicitement conçu- mais aussi pour la ou les phases ultérieures de génération de code à partir d'un langage intermédiaire.

Dans tous les cas, cette pratique reste empreinte d'un certain degré de ''bricolage'', car l'outil initial avait d'autres objectifs.

8 6 2 Le langage SNOBOL

Par contre, le langage SNOBOL a dès le départ été conçu pour introduire de manière propre l'analyse syntaxique comme primitive de base de programmation.

Une étude complète du langage et de son implémentation est faite en [93].

Ce langage permet:

- de définir des grammaires (appelées en l'occurence MODELES)

- d'analyser des données selon ces modèles

- d'effectuer des actions sémantiques lors de cette analyse

Ceci est très proche des besoins de programmation de nos filtres ...

La principale différence vient de ce que le filtrage tel que nous l'envisageons doit respecter des contraintes ''PHYSIQUES'':

- pas de retour arrière (contrainte spatiale)

- travail au vol (contrainte temporelle)

Au contraire, SNOBOL travaille dans un univers purement logique (logiciel), et le but recherché n'est pas une performance (quantitative), mais un agrément (qualitatif) dans l'utilisation du langage.

Nous exposons maintenant de manière informelle les principaux dispositifs de SNOBOL liés à l'utilisation de l'analyse syntaxique. Une étude intéressante consisterait à examiner lesquels, parmi ces dispositifs, seraient réalisables en respectant les contraintes quantitatives du filtrage, pour aboutir à la notion de ''machine SNOBOL''.

Dans SNOBOL, il existe un type de données particulier appelé MODELE, sur lequel sont définies des opérations analogues à nos EST, OU, PUIS et PLUSIEURS du langage LS, et que permettent de définir des langages algébriques.

La différence est que, dans le cas de SNOBOL, les opérations sur les modèles peuvent être effectuées à tout moment lors de l'exécution d'un programme, tout comme des opérations arithmétiques.

Etant donné une variable M de type modèle, on peut faire exécuter l'analyse d'une autre variable de type chaine de caractère C par ce modèle.

EXEMPLE:

```
M = ('AA'| 'BB') 'CC'

C   M    :S (OK) :F(KO)

OK ......
.......
KO .....
```

.....

Ce programme calcule d'abord le modèle M, puis l'applique à C. Si C satisfait à ce modèle, l'exécution se poursuit à l'étiquette OK du programme, sinon à l'étiquette KO.

Le résultat de l'analyse sert ici à provoquer des branchements conditionnels.

On peut de plus insérer des intructions d'affectation à l'intérieur d'un modèle:

Exemple:

```
BASE = ('HEX' | 'DEC' | AUTRE) . VAR
```

Ceci définit le non-terminal BASE, et spécifie de plus que, lors de l'analyse, la partie du texte correspondant à ce non-terminal sera affectée à la variable VAR.

L'analyse syntaxique se traduira ainsi par un certain nombre d'EFFETS DE BORD.

Une autre généralisation importante est que l'on peut définir des PROCEDURES rendant un résultat de type MODELE, ce qui permet d'analyser des langages plus généraux que les langages algébriques.
Par exemple, il existe des procédures prédéclarées comme TAB(n) et RTAB(n) qui acceptent les n premiers (respectivement derniers) caractères de la chaine en cours d'analyse s'ils existent.

Il n'existe aucune notion de précompilation des modèles. Toute l'exécution est interprétée. Par exemple, dans:

```
C    ( PROC (I,J) ) | (PROC1 (K,L) )
```

l'analyse de la chaine C par le modèle situé à sa droite va d'abord provoquer l'appel et l'évaluation de la procédure PROC qui va rendre un résultat de type modèle, mais peut en plus affecter des variables qui, par effet de bord, interviendront dans l'évaluation de la procédure PROC1 qui suit, soit en paramètres, soit en tant que variables globales.

De plus, les procédures SNOBOL peuvent être récursives.

Les possibilités offertes par SNOBOL sont donc très grandes, sans doute un peu trop, si l'on veut s'astreindre à une programmation bien construite.

N.B. A coté de ces dispositifs spécifiquement dérivés de l'analyse syntaxique, SNOBOL dispose de structures de données et de controle plus classiques.

CHAPITRE 9

UNE TECHNIQUE D'INTERPRETATION DE APL EXTENSIBLE A L'ALGEBRE RELATIONNELLE

9 UNE TECHNIQUE D'INTERPRETATION DE APL EXTENSIBLE A L'ALGEBRE RELATIONNELLE

Il existe de nombreuses ressemblances entre deux langages encore un peu marginaux, APL [87] et l'algèbre relationnelle:(AR en bref) [88].

- tous deux sont des ALGEBRES sur des ENSEMBLES, la notion d'ensemble étant complètement supportée tant au niveau des données que des opérateurs.

- tous deux sont des langages très ''propres'', issus de travaux de mathématiciens, et faisant l'objet de continuelles études formelles. [75], [74], [92]

- tous deux sont à la base des langages non procéduraux et conversationnels

- ils connaissent un succès grandissant

Par contre, ils ont quelques caractéristiques qui les opposent:

- APL manipule ses TABLEAUX par ADRESSES,

- AR manipule ses RELATIONS par ASSOCIATIVITE

- APL est très riche en opérateurs de CALCUL

- AR est surtout orienté vers L'ACCES aux informations

Dans le cadre de cette thèse consacrée au traitement des ENSEMBLES DE DONNEES, il nous a semblé intéressant de consacrer un chapitre à une tentative de rapprochement, sinon de fusion entre ces deux langages. Il est clair qu'un langage qui réunirait leurs qualités, (avec si possible en plus un zeste de ''types

abstraits'') constituerait un progrès fondamental dans le domaine du génie logiciel.

Tout en voulant rester bref, ce chapitre essaie d'aller au fond des choses, en se plaçant au niveau des mécanismes d'interprétation (si possible optimisée) de tels langages, plutot qu'à celui de leur définition formelle.

Nous allons en conséquence successivement étudier une méthode d'interprétation simple mais optimisée de APL et indiquer brièvement comment elle peut s'étendre à AR.

9 1 PARTICULARITES ET PROBLEMES DU LANGAGE APL

Rappelons quelques points caractéristiques du langage APL, utiles pour comprendre les exemples qui suivront.

APL diffère de la famille de langages comme FORTRAN ou PASCAL sur beaucoup de points importants:

-la liaison entre les noms des objets, les valeurs et les types de ces objets n'est effectuée qu'à l'exécution, et ne peut pas être compilée

-les variables designent des ENSEMBLES (précisément des tableaux à plusieurs dimensions), et les opérateurs opèrent globalement sur ces tableaux, sans qu' il ne soit nécessaire d'accéder individuellement aux éléments.
Par exemple, si A,B,C sont des matrices,l'instruction:

C←(3 2 ↑ B) + A

signifie:
faire la somme de la matrice A avec une matrice constituée des 3 premières lignes lignes de B, limitées aux deux premières colonnes, et ranger le résultat dans C.

En FORTRAN, un programme équivalent serait:

```
      DO 10 I= 1, 3
      DO 20 J= 1,2
      C(J,I) = B(J,I) + A(J,I)
   20 CONTINUE
   10 CONTINUE
```

Plus gènèralement, il existe de nombreux opèrateurs complexes et sophistiquès en APL qui n'ont pas d'èquivalent proche ou lointain dans les langages comme FORTRAN ou PASCAL.En particulier, certains opèrateurs concernent les structures des donnèes plutot que leurs valeurs. Exemple:

```
      ⍉ A,[2] B
```

Si A et B sont des matrices, cette expression les concatène le long de leurs colonnes, (seconde dimension) puis prend la transposèe du rèsultat.

Si A ↔ 1 2
       3 4

```
 B ↔ 5 6 7
     8 9 10
```

alors le rèsultat de l'expression ci-dessus est:

```
  1 3
  2 4
  5 8
  6 9
  7 10
```

Un autre mècanisme puissant en APL est l'indexation:

si A, B, C sont des vecteurs, alors dans:

```
  B[J] + A[I]
```

I et J peuvent avoir n'importe quelle dimension, et n'être pas seulement des scalaires ni même des vecteurs.Par exemple, si:

```
B ↔ 5 2 8 4
A ↔ 10 20
J ↔ 1 2
      2 3
I ↔ 1 1
      2 1
```

alors B[J] + A[I] ↔ 15 12
22 18

De plus, on peut indexer non seulement des variables mais aussi des expressions:

C← (A+B) [I]

et naturellement, une expression indexée peut être indexée a son tour:

((A[I]+B)[J+K]+E)[F]

Pour les tableaux multidimensionnels, la syntaxe est la suivante:

A[I;J;K;L]

Mais le nombre de dimensions de A[I;J;K;L] n'est pas nécessairement de 4, c'est la somme des nombres des dimensions des indices I, J, K, L.

Il y a des dizaines d'opérateurs en APL. Finalement, APL peut être considéré comme une ALGEBRE SUR LES TABLEAUX.

La programmation et la mise au point en APL est une activité très agréable.On considère que cela prend entre 5 et 10 fois moins de temps qu'en FORTRAN par exemple.

Bien qu'APL présente de petits et de grands défauts, les chiffres ci-dessus devraient inciter la communauté informatique à s'intéresser de plus près et avec moins d'arrière -pensées à APL. L'utilisation de APL semble parfois assimilée a une 'tricherie'

En fait, le principal désavantage de APL est que les implémentations actuelles conduisent à des temps d'exécution très longs, si on les compare au temps des mêmes problèmes programmés en PASCAL ou FORTRAN.

La raison est simple:

PASCAL et FORTRAN ont été conçus pour être compilés sur des machines existantes, et APL a été conçu pour décrire et résoudre des problèmes, sans idée d'exécution ultérieure.''LA'' machine APL n'a pas encore vu le jour .....

Ce chapitre est un petit pas vers cet objectif.Nous espérons convaincre le lecteur que, dans de nombreux cas, une machine APL peut être plus efficace q'une machine PASCAL ou FORTRAN (donc a fortiori qu'une machine universelle) ,et ceci pour le même problème et avec la même technologie.

Plus précisément, nous voulons démontrer qu' il est encore plus intéressant de faire une machine spécialisée pour APL que pour FORTRAN ou PASCAL. Et ceci est particulièrement réjouissant pour les circuits intégrés: ils auront encore quelque chose à intégrer après avoir digéré le langage IBM 370 !

9 2 TECHNIQUES D'INTERPRETATION DE APL

41 APL n'est pas compilable au sens habituel du terme

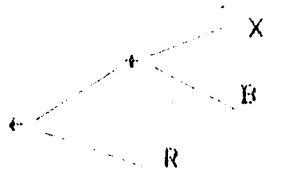
Considérons la fonction APL suivante:

```
  ∇R←A FONC B;X
[1] X←5
 .
 .
[7] X← 3 2 ⍴0
 .
 .
[10] R←X + A + Y Z T
 .
 .
[15] R←X + B
∇
```

R est le résultat retourné par la fonction.
A et B sont les deux paramètres d'entrée
X est une variable locale

Il n'y a pas de déclarations en APL.Il est impossible de connaitre le type (entier, flottant, booléen, ...) ni le nombre de dimensions de A et B. De plus, ces caractéristiques peuvent varier d'un appel à l'autre ...

Il en est de même pour X.Il est donc impossible de compiler par exemple la ligne [15] dans un code machine classique, puisque rien n'est connu sur les opérandes.La seule chose connue au niveau du source de la ligne [15] est sa structure, encore appelée parfois son 'arbre abstrait', bien que ce soit la seule chose 'solide' ...



Malheureusement, ceci n'est pas vrai en général:
Dans la ligne [10] : Y Z T
peut se comprendre comme

(Y Z T)

si Z est une fonction dyadique
ou comme

(Y (Z T))

si Y et Z sont monadiques.

Et tout ceci sera découvert seulement à chaque exécution de la ligne, et peut changer à l'exécution suivante, puisqu'il existe des opérateurs qui permettent de créer, supprimer, modifier dynamiquement des fonctions ....

Ces possibilités sont évidemment parmi les parties les plus controversées d'APL. Pour beaucoup d'informaticiens, elles sont des raisons suffisantes pour rejeter définitivement ce langage.En fait, elles ne sont que rarement utilisées, mais, quand elles s'avèrent utiles, elle épargnent des semaines ou des mois de

programmation, puisqu'elles n'ont pas d'équivalent dans les autres langages.
Quoi qu'il en soit, elles sont partie intégrante d'APL, et doivent être prises en compte.

Il y a deux attitudes possibles face à ce problème de non compilabilité:

1)- réinterpréter intégralement à chaque exécution

2)- compiler la ligne lors de sa première exécution
- conserver le code objet
- conserver le contexte de compilation (arbre abstrait et nature des variables)
- à la prochaine exécution, si le contexte n'a pas changé, exécuter l'ancien code, sinon recompiler

9 3 L'INTERPRETATION DE APL EST OPTIMISABLE

Puisque APL est un langage très riche, il y a plusieurs manières d'exécuter une expression donnée, donc possibilité d'OPTIMISER cette exécution en fonction de critères donnés.

Exemple:
(A+B)[I] où A, B, I sont des vecteurs
Une solution est de

a)- évaluer A+B
b)- indexer le résultat par I

c'est à dire de faire:

```
W←A+B
W[I]
```

Cette méthode est généralement appelée la méthode naive. Elle est utilisée par la plupart des systèmes APL disponibles sur le marché. Le principe est d'évaluer l'arbre abstrait des feuilles

vers la racine, comme une expression postfixée.Cela correspond d'autre part à la sémantique 'officielle' d'APL.

Cette technique est très simple, mais le problème est que si A et B ont 10 éléments et I seulement 2, alors on exécute 10 additions pour ne conserver que deux résultats ...

Pour éviter ce gachis, les méthodes d'optimisation consistent généralement à faire:

(A+B)[I] → A[I] + B[I]

Cet exemple est très simple à comprendre, mais les choses se compliquent avec des opérateurs tels la concaténation, les rotations, transpositions ...

La meilleure et la plus complète des méthodes d'optimisation connues à ce jour est probablement celle proposée par TUSERA [69].

Elle consiste à compiler chaque ligne avant chacune de ses exécutions, puis à exécuter le code objet.

Puisque le compilateur connait toute la ligne, il est capable d'optimiser GLOBALEMENT sa compilation, au contraire de l'interprétation naïve qui progresse aveuglément des feuilles de l'arbre vers la racine sans savoir où elle va.

Le code objet comporte autant de boucles d'itération qu'il y a de dimensions dans le résultat de l'expression à calculer. Chaque passage dans la boucle interne calcule un élément du résultat.

Les calculs d'indices de boucles sont optimisés ainsi que l'allocation des registres d'index.

Le prix à payer est cette compilation permanente, qui de plus est optimisée, donc longue.

Comme cela est exptiqué dans [70], il est évident que cette 'méthode radicale ' ne commence à payer que si les variables sont assez grandes, c'est à dire si il y a assez d'itérations dans le code objet pour amortir le temps passé à le compiler.

Comme cela a été mentionné plus haut, cette méthode peut être améliorée si l'on essaie de conserver le code objet d'une exécution à une autre après avoir vérifié qu'il est toujours valable.

Note importante

La transformation de (A+B)[I] en A[I] +B[I] n'est malheureusement pas toujours optimale: par exemple si ce sont A et B qui ont deux éléments et I qui en a dix ...

Ainsi le code objet compilé peut ne pas être optimal, même si il est correct. Il faut considérer non seulement le nombre de dimensions mais aussi leur taille, chose qui en APL varie beaucoup plus fréquemment, puisque cela correspond au fait d'utiliser les possibilités de gestion IMPLICITE de la mémoire dynamique .

En conséquence, il est clair que une bonne méthode devrait permettre un très haut niveau de flexibilité au moment de l'exécution, tout en essayant de minimiser le temps passé a CALCULER la stratégie optimale d'évaluation.

Le paragraphe qui suit propose une nouvelle méthode d'évaluation qui essaie de satisfaire ces objectifs.

9 4 PROPOSITION D'UNE METHODE D'INTERPRETATION

Notre but ici n'est pas de décrire un interpréteur complet, mais de proposer une technique nouvelle d'inteprétation, et d'en montrer l'applicabilité à quelques opérateurs APL, en particulier au plus important, l'indexation, duquel se déduisent la plupart des autres.

9 4 1 Comment compiler la syntaxe

L'unité de base d'un programme APL est la LIGNE. Les lignes sont numérotées et tout branchement spécifie un numéro de ligne.Une ligne est une expression qui peut contenir des affectations et des appels de fonctions.

Exemples:

```
1 + 1
A← 1+2
A← (B←C-3) +A
A[I←I+1]← X FONC Y FONC1 Z
```

Puisque les paramètres sont passés par valeur, une ligne constituée ainsi:

```
... ( expr1 FONC expr2) ...
```

peut toujours etre remplacée par:

```
W1←expr1
W2←expr2
R←W1 FONC W2
```

... R ...

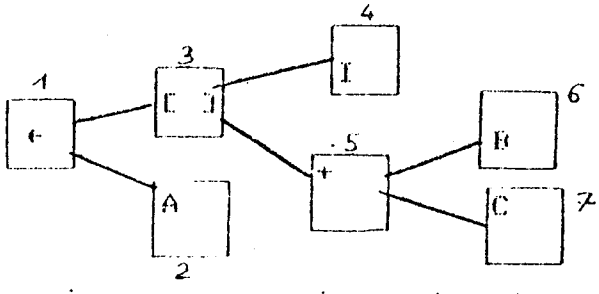
En appliquant ce type de transformations chaque fois que possible, on aboutit à une syntaxe que nous appellerons NORMALE où une ligne est soit:
- un appel d'une fonction sans expressions
- une expression sans appel de fonctions

Mais un même nom peut représenter des choses différentes au cours du temps: une variable, une fonction à 0 1 ou 2 paramètres et à 0 ou 1 résultat. Il faut donc adopter la technique suivante: la première fois que la ligne est exécutée, elle est transformée comme mentionné ci-dessus. Les fois suivantes, il suffit de tester si les variables sont toujours des variables et si les fonctions sont toujours des fonctions et de même nature. Cette vérification est assez simple puisqu'elle s'effectue au niveau des lignes normales dont chacune contient au plus une fonction par construction. D'autre part, de tels changements sont intuitivement extrêmement rares au cours d'une session de travail, et cette vérification peut consister en un premier test rapide d'un indicateur précisant la date de la dernière modification de ce type, qu'il suffit de comparer à la date de la précédente décomposition de la ligne. Quand un changement est détecté, il faut reprendre la décomposition de TOUTE la ligne source.

Dans le paragraphe qui suit, nous ne considérons que des lignes constituées de pures expressions.

9 4 2 Un exemple simple

Représentons une expression par son arbre abstrait:
A←(B + C) [I]



Imaginons que ceci représente un réseau où chaque boite est un ordinateur. Ceux ci communiquent entre eux en recevant et envoyant des messages le long des arêtes de l'arbre.

Alors, l'interprètation de cette expression peut se dèrouler de la manière suivante:

**- 1 demande à 2 : envoie moi une adresse, soit (a)
**- 1 demande à 3 : envoie moi une valeur, soit (b)
**- quand 1 a reçu (a) et (b), il range (b) à l'adresse (a)

**- le noeud 3 recoit de 1 'donne-moi une valeur '. Alors il demande a 4 'donne-moi une valeur' , soit (i) cette valeur.
**- quand il recoit (i), il demande à 5 'donnez moi votre (i) EME valeur'
**- alors 5 demande à 6 et à 7 'donnez moi votre (i) EME valeur'.
**- 6 et 7, qui sont des variables, envoient leur (i) éme valeur à 5, qui en fait la somme et la retoutrne à 3 qui à son tour la retourne à 1 ...
**- 1 demande alors à 2 son adresse suivante et à 3 sa valeur suivante, et tout recommence, jusqu'à ce que 2 et 3 disent : je vous ai tout envoyè. Alors l'opèration est terminèe.

Plutot que de considèrer les noeuds comme des ordinateurs, on peut aussi les considèrer comme des processus ou des coroutines èchangeant des messages, et exècutès sur un même processeur.

Le problème de l'optimisation de (B+C)[I] quand il y a moins d'èlèments dans I que dans B et C est alors facilement rèsolu: seuls les indices nècessaires sont èvaluès. Cette mèthode est à la base une mèthode optimisante.

Elle est à rapprocher fortement de la mèthode de MARCHAL [71], pour qui chaque noeud est un sous-programme qui appelle les les sous-programmes situès à sa droite, et où les informations sont transmises par passage et retour de paramètres.

Il est important de savoir que la mèthode [69] est nèe d'une critique de la mèthode [71], en particulier de la lourdeur du mècanisme d'appel de sous programmes et de transmissions d'informations. Partant de notre modèle ''naif'' d'ordinateurs communiquant entre eux, notre but est prècisèment de rèaliser un bon compromis entre les deux mèthodes:

- tendre vers les performances de la compilation ...

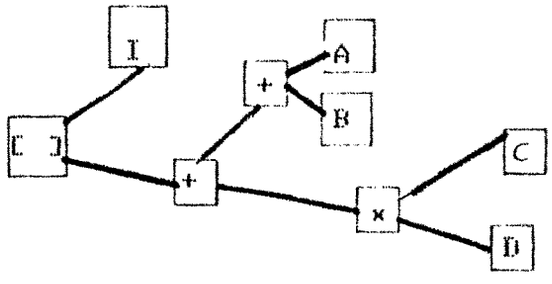
- ...avec la simplicité de l'interprétation

- un troisième objectif sous-jacent est d'utiliser une technique suffisamment souple pour permettre des EXTENSIONS à APL, sans remettre en cause la méthode d'interprétation.

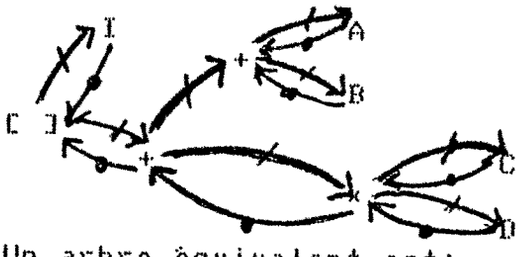
Examinons les coûts de notre méthode d'optimisation. Ils consistent en:

-passation de controle entre les noeuds et changements de contextes
-échange de messages
Essayons de minimiser ces points en raisonnant sur un exemple:

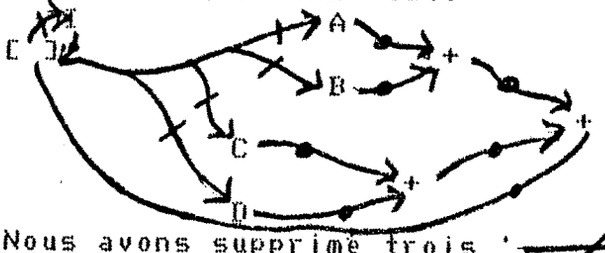
Soit l'expression: ((A + B) + (C × D)) [I]
Son arbre est:



Explicitons les liens de la manière suivante:
⟶ signifie 'demander une valeur'
⟶ signifie 'retourner une valeur'



Un arbre équivalent est:



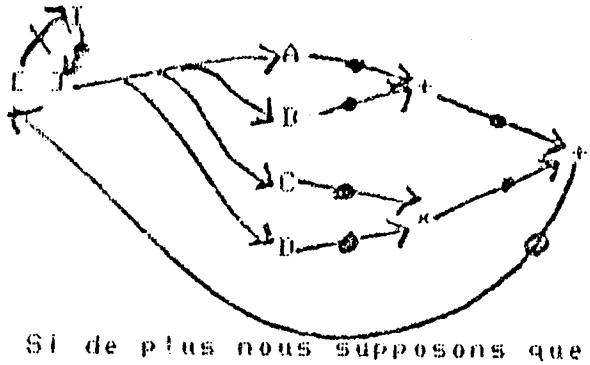
Nous avons supprimé trois '⟶' et en conséquence trois entrées et sorties pour les noeuds '+' et '×'

De plus, on voit que A,B,C,D reçoivent la MEME valeur d'index depuis '[ ]'

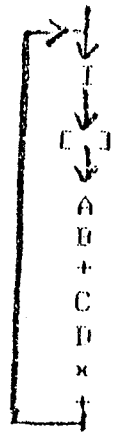
Au lieu d'envoyer explicitement cette valeur quatre fois, on pourrait la ranger dans un endroit accessible à A,B,C,D et simplement envoyer le CONTROLE (sans valeur) à ces noeuds. Nous venons de réinventer le registre d'index! ... si nous supposons que toute variable est indicée implicitement.

Si nous utilisons le symbole '⟶' pour signifier

'donner le controle', alors le graphe devient:



Si de plus nous supposons que nous travaillons sur un monoprocesseur, , avec des opérateurs travaillant sur une pile d'évaluation, alors le graphe peut être représenté de la manière suivante, qui n'est autre qu'une forme postfixée:



Et nous retombons sur un cas de figure très proche de ce que la méthode [69] obtient par compilation.

Mais nous verrons dans la suite que notre graphe va être indépendant du nombre de dimensions des variables, contrairement à la méthode de compilation pure.

9 5 TRAITEMENT DU CAS GENERAL

L'exemple précédent était très simple.

Il montre cependant que le cout de transport des indices dans la méthode [71] peut être fortement réduit si:

1)- on ne les passe pas en paramétres de procedures, mais par l'intermédiaire de registres

2)- on représente l'expression de manière postfixée et non sous forme d'un arbre 'naif' avec pointeurs

Nous allons maintenant aborder le problème dans sa généralité:

a) une expression indexée peut à son tour être indexée

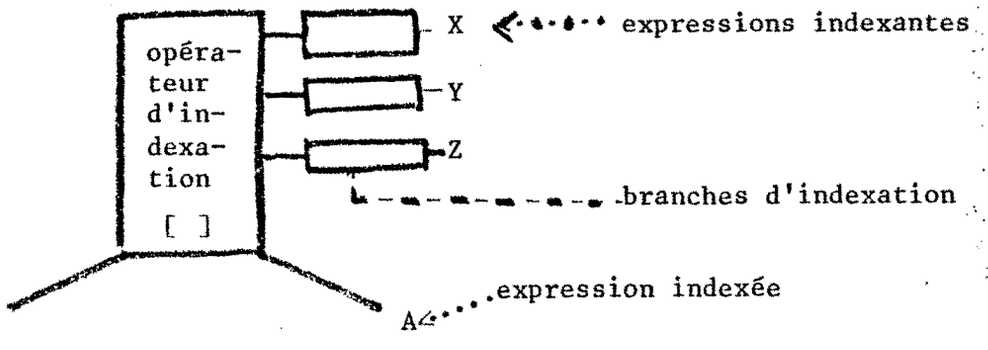
b) les tableaux peuvent avoir plus d'une dimension

Pour résoudre a) nous aurons besoin d'une PILE de registres d'index appelée dans la suite PIX

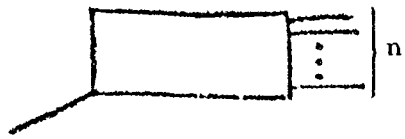
Pour résoudre b), il devra y avoir au sommet de cette pile d'index autant de registres que de dimensions dans l'expression courante.

9 5 1 ANALYSE DE L'INDEXATION

L'indiciation multiple, comme dans A[X;Y;Z] sera représentée graphiquement par:



De même, une variable à n dimensions sera représentée ainsi:



Rappel:

En APL, le nombre de dimensions d'une expression indexée est égale a la somme des dimensions des expressions indexantes.

Exemple:

Si M1 et M2 sont des matrices et V1 et V2 des vecteurs, alors:

V1[M1] a deux dimensions
M1[V1;V2] a deux dimensions
M1[M2;V1] a trois dimensions
M1[M1;M2] a quatre dimensions

Par exemple, si T← M1[M2;V2]
alors T[I;J;K] ↔ M1[M2[I;J];V2[K]]

Considérons maintenant l'exemple suivant:

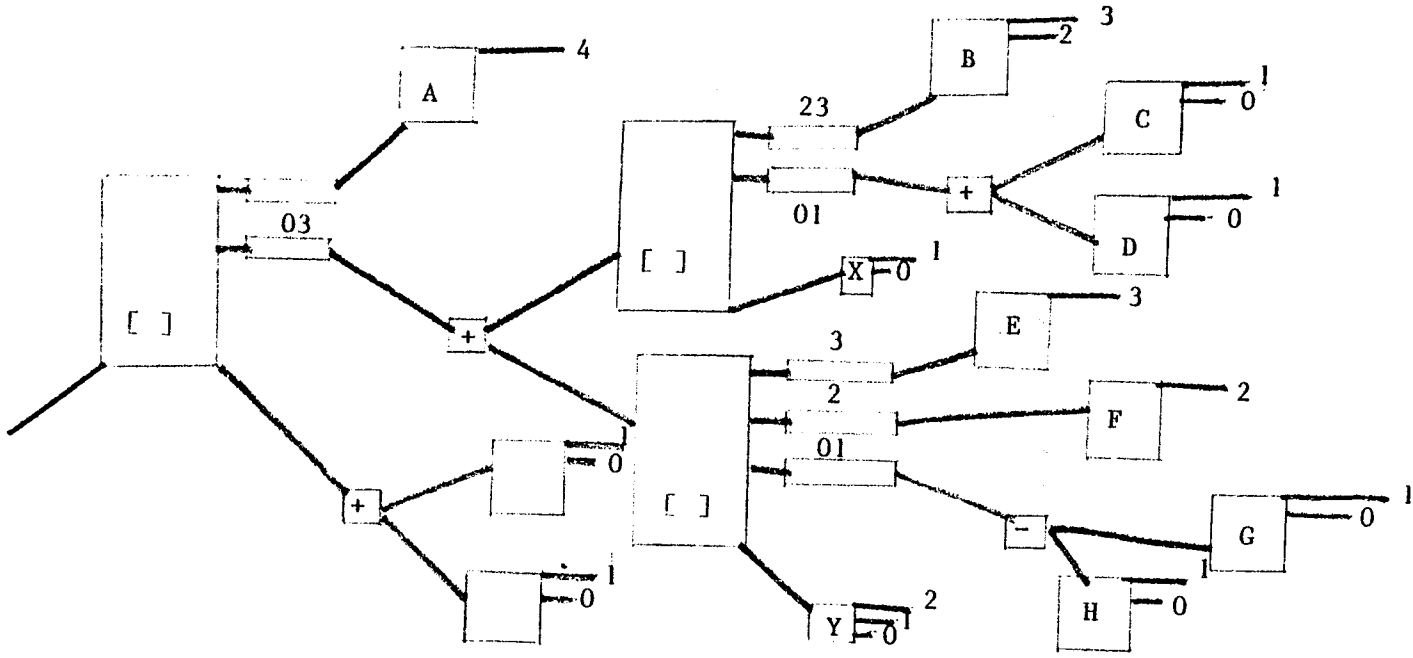
(M+N)[A;X[B;C+D]+Y[E;F;G-H]]

où M, N, H, G, D, C et B ont deux dimensions

Y a trois dimensions

A, E et F ont une dimension

Nous le représenterons ainsi:



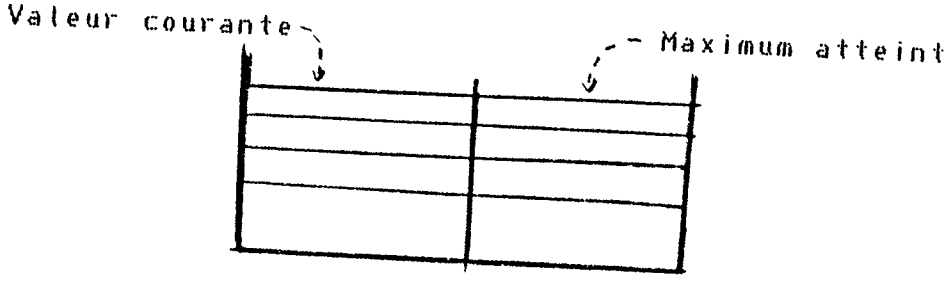
Pour chaque variable, les nombres à droite indiquent les numéros de registres du sommet de pile d'index qu'elle doit utiliser. Le nombre de ces registres est évidemment égal au nombre de dimensions de la variable.

Nous appellerons ces numéros les Segments d'Indices (SI) associés à une variable. Nous associerons de même un SI à chaque branche d'indexation.

Le problème est: comment calculer ces numéros à partir de la connaissance de l'arbre et des dimensions des variables ?

Remarquons d'abord que ce problème est limité au sommet courant de la pile d'index, c'est à dire qu'il est local à chaque opérateur ' [ ] '. Il sera évidemment aussi résolu par une pile appropriée.

Nous utilisons une pile à deux éléments:



Voici l'algorithme:

0) La pile est initialisée à (-1,-1)

1) Pour une variable:

* quand elle reçoit le contrôle, faire:
Max atteint ← valeur courante
Pour chacune de ses dimensions, faire
incrémenter Max atteint
le résultat de cette incrémentation donne le numéro du segment d'indice associé à la dimension courante.

2) Pour une branche d'indexation:

* passer le contrôle à l'expression indexante

* quand le contrôle revient, faire

SI de la branche ← (Valeur courante, Max atteint)

Valeur courante ← Max atteint

* rendre le contrôle à l'opérateur d'indexation

3) Pour un opérateur d'indexation:

* quand il reçoit le contrôle, empiler le sommet de la pile (duplication du sommet)

* faire Max atteint ← valeur courante

* donner le contrôle à ses branches d'indexation, du bas vers le haut

* quand la dernière branche retourne le controle, la pile a l'allure suivante:

Nouvelle val cour	nouveau Max atteint
Ancienne val cour	ancien Max atteint

Alors on la transforme en

-1	-1
Ancienne val cour	ancien max atteint

* donner le contrôle à l'expression indexée

* quand le contrôle revient, dépiler le sommet de pile, rendre le controle à l'appelant

Une fois cet algorithme exécuté, chaque variable et chaque branche d'indexation connait le numéro des indices qui la concernent.

9 5 2 REGLES D'INTERPRETATION

Revenons maintenant à l'exécution de l'expression.

Les règles de base sont les suivantes:

1)- Lorsque le controle est donné à un noeud de l'arbre, le sommet de la PIX contient les coordonnées de l'élément demandé à ce noeud.

2)- Les opérations de calcul (+,× ,...) NE TOUCHENT PAS à PIX. Ils se contentent de demander, recevoir et transformer les valeurs de même indice de leurs paramétres.

3)- Les opérateurs de structuration (indexation, rotation, transposition, ...) vont MODIFIER LA VALEUR DE PIX avant de passer le controle à leurs opérandes.

4)- Les feuilles de l'expression (les variables) trouvent au sommet de PIX les valeurs leur servant à calculer l'adresse physique de l'élément qui leur est demandé et retournent la valeur de cet élément.

Notre but ici n'est pas de décrire un interprèteur complet, mais seulement d'illustrer la méthode sur quelques opérateurs.

9 5 2 1 Premier exemple: L'INDEXATION.

Quand il reçoit le controle, il effectue les opérations suivantes:

1)- Appeler successivement toutes ses branches d'indexation. Chacune lui retourne une valeur d'index

2)- Empiler toutes ces valeurs reçues sur PIX

3)- Appeler son expression indicée qui lui retourne un résultat

4)- Dépiler PIX

5) rendre le controle à son appelant en retournant le résultat reçu en 3)

9 5 2 2 Deuxième exemple: La TRANSPOSITION

V ⍉ M

M est l'expression transposée et V le vecteur de transposition.

Quand il reçoit le controle, il exécute les opérations suivantes:

0)- Acquérir la valeur V de son paramètre gauche

1)- Empiler un nouveau sommet de pile égal à l'ancien indicé par V (au sens de l'indiciation des vecteurs en APL)

2)- Appeler son paramètre droit et en recevoir une valeur

3)- dépiler PIX

4)- rendre le controle à l'appelant en retournant la valeur reçue en 2)

Par rapport à la méthode [69], l'avantage est qu'il n'est pas nécessaire de transformer l'arbre de l'expression, ni de compiler ensuite l'arbre transformé en un code objet dépendant des dimensions des variables.

9 5 3 OPTIMISATION DES CALCULS D'INDICES

Cette méthode a néanmoins un inconvénient de [71] supprimé par [69]:

en général, entre deux valeurs successives du sommet de pile, seuls les indices de poids faible auront changé.

Par exemple, dans l'expression M[1 2 3; 1 2; 1 2 3 4], la pile d'index prend successivement les valeurs suivantes:

1 1 1 / 1 1 2 / 1 1 3 / 1 1 4 / 1 2 1 / 1 2 2 ... etc ...

D'une part les indexants d'indices supérieurs recommencent plusieurs fois le même calcul pour donner les mêmes valeurs d'indices, d'autre part, les indexés refont une partie de leur calcul d'indices avec les mêmes opérandes pour le même résultat.

La méthode [69] consistant à compiler l'expression sous la forme de n boucles imbriquées si il y a n dimensions, ce problème est résolu: on reste dans la boucle la plus interne (celle de la dimension qui varie le plus vite) aussi longtemps que les valeurs calculées par les boucles englobantes restent valables.
Exemple:

```
  M[ A;B;C]
avec A ↔ 1 2 1
     B ↔ 1 2
     C ↔ 2 1
```

Un '' équivalent FORTRAN '' de notre méthode est:

```
DO I = 0 TO 2
DO J = 0 TO 1
DO K = 0 TO 1
ADRESSE DANS M ← (A[I] × 4) + (B[J] × 2) + C[K]
END
END
END
```

Le code [69] est:

```
DO I = 0 TO 2
ADR ← A[I] × 4
DO J = 0 TO 1
ADR ← ADR + B[J] × 2
DO K = 0 TO 1
ADRESSE DANS M ← ADR + C[K]
END
END
END
```

L'optimisation a consisté à sortir les invariants des boucles.

La seconde méthode nécessite 9 multiplications contre 24 pour la première.

Voici une solution pour retrouver cette optimisation dans notre méthode.

Plutot que de transmettre à chaque fois TOUS les indices d'un élément au sommet de PIX, on ne transmet que ceux qui ont changé depuis la dernière fois.

Le sommet de PIX contient donc des doublets de la forme:

(Numero de dimension ayant changé, nouvelle valeur)

A partir de ces nouvelles valeurs, les opérateurs qui modifient PIX vont à leur tour calculer d'autres nouvelles valeurs pour leurs paramètres, mais toujours seulement celles ayant changé par rapport à la dernière fois.

Par exemple, la transposition V&M empilera un nouveau sommet contenant tous les couples (d,v) tels que le couple (V[d],v) appartient au sommet courant.

Quand aux variables, leur calcul d'adresse se simplifie.
Soit AP l'adresse précédente calculée avec les valeurs d'indices précédentes IPi. Soit PASi le pas de la iéme dimension.

Alors l'adresse courante AC se calcule en fonction des nouveaux indices courants ICj selon la formule:

$$AC = AP + \text{Somme sur } j \text{ de } PASj \times (ICj - IPj)$$

Au lieu de faire autant de multiplications qu'il y a de dimensions, on en effectue un nombre égal au nombre d'indices qui ont changé.

9 6 PASSAGE DES TABLEAUX AUX RELATIONS

Plaçons-nous au niveau du mècanisme d'interprètation pour examiner comment il serait adaptable au traitement de RELATIONS plutot que de tableaux.

Une relation n-aire est un ensemble de n-uplets.

Supposons que les èlèments de la pile PIX ne soient plus des indices mais des n-uplets; lorsque l'on donne le controle à un sous-arbre, on lui demande de retourner en rèsultat le n-uple dont la valeur est celle du sommet de PIX, SI IL EXISTE.

Une relation n'est pas une structure complète comme l'est un tableau (qui est un PAVE dans R**n).

On ne peut ènumèrer a priori les n-uples. Il faut plutot, pour chaque domaine i, donner non pas une valeur mais un INTERVALLE DE VALEURS (INFi, SUPi).

Le sommet de PIX contient un tel intervalle pour chaque domaine i, et il signifie:

Retourner TOUS les n-uples (x1, ,xn) tels que:

(INF1 ≤ x1 ≤ SUP1) ET (INF2 ≤ x2 ≤ SUP2) ... ET (INFn ≤ xn ≤ SUPn)

N.B. Si INFi = SUPi pour tout i, on se retrouve dans le cas de APL.

De plus, il doit exister des valeurs rèservèes de INFi (resp SUPi) qui dèsignent le minimum (resp le maximum) absolu des valeurs du domaine i, et qui correspondent à l'absence totale ou partielle de conditions sur ce domaine.

Remarque fondamentale: on voit qu'à UN sommet de pile de PIX peuvent correspondre PLUSIEURS résultats retournés. Ceci est une différence de base avec APL, qui fait en particulier que la solution [69] (compilation de boucles) pour l'interprétation de APL n'est pas extensible au cas des relations, alors que la notre (processus asynchrones communicants) l'est.

Toute condition de SELECTION de l'algèbre relationnelle peut se mettre sous forme normale disjonctive, c'est à dire sous forme d'un 'OU' de monomes dont chacun a l'allure d'un sommet de PIX tel qu'il vient d'être défini.

Pour réaliser la SELECTION, il suffit donc d'appeler successivement l'expression à sélecter avec à chaque fois un monome en sommet de PIX.

Les valeurs retournées ne sont pas comme en APL des scalaires mais des n-uplets. Il faut disposer d'un opérateur supplémentaire qui réalise la PROJECTION de l'algèbre relationnelle, c'est à dire qui précise le sous-ensemble de domaines qui nous intéressent. Ce sous-ensemble devra aussi être indiqué dans chaque élément de PIX.

Nous allons maintenant généraliser l'INDEXATION de APL pour aboutir à la notion de JONCTION de l'algèbre relationnelle.

Etant données deux relations R1 et R2, on appelle JONCTION sur égalité R1 [i = j] R2 la relation dont les éléments sont la concaténation des couples d'éléments de R1 et R2 tels que le iéme domaine de R1 soit égal au jème de R2. De plus, cette valeur commune n'est gardée qu'une seule fois.

Par exemple, si R1 et R2 sont deux relations binaires, et se (a,b) ∈ R1 et (b,c) ∈ R2, alors (a,b,c) ∈ R1 [2 = 1] R2.

Ceci ressemble beaucoup à la situation suivante en APL:

T1 et T2 sont deux tableaux à 2 dimensions. T ← T1[; T2] est un tableau à trois dimensions, tel que T[a;b;c] est égal à T1[a;T2[b;c]].

D'où l'idée d'étendre ainsi l'indexation APL aux relations:

Etant données deux relations R1 et R2, on définit

R1[; ... ; j :: R2 ; ... ;]

iéme coordonnée

comme égal à la jonction sur égalité R1 [i = j] R2

Au delà de cette définition, la chose remarquable est que l'algorithme d'interprétation de l'indexation que nous avons donné pour l'indexation de APL est toujours valable avec des relations. La seule chose nouvelle est que, à chaque branche d'indexation, il faut indiquer le numéro du domaine de l'expression indiçante que l'on doit empiler au sommet de PIX. (C'est le numéro qui est à gauche de ''::'' dans la syntaxe que nous avons donnée à l'indexation).

Ce numéro est inutile en APL, puisque l'expression indicée ne retourne toujours qu'un scalaire, alors qu'ici, elle retourne un n-uple.

CONCLUSION

CONCLUSION

Cette thèse a décrit un travail en ''architecture d'ordinateurs''. Partant d'un domaine d'applications précis -la gestion des ensembles de données de grande taille résidant sur des supports séquentiels - nous avons suivi une démarche qui nous semble de portée assez générale, et qui peut s'énoncer ainsi:

1)- phase de RESTRICTION de la généralité du problème

2)- phases de CONCEPTION DESCENDANTE, se décomposant en:

2.1)- descente qualitative: choix d'algorithmes et de structures de données

2.2)- descente quantitative statique: évaluation statique ''à la compilation'' des algorithmes et des structures.

2.3)- descente quantitative dynamique: évaluation dynamique ''à l'exécution'' des algorithmes et structures

3) phases DE CONCEPTION ASCENDANTE:

3.1) écriture d'un logiciel de programmation de l'architecture obtenue en fin de descente

3.2) recherche d'extensions du domaine d'application de l'architecture n'entraînant que des modifications minimes

Rappelons le contenu de ces étapes dans notre cas:

1)- restriction de la généralité du problème.

Plutot que d'attaquer de front la conception d'une ''machine bases de données '', nous nous sommes limités aux opérations de CONSULTATION d'UN SEUL FICHIER, eu égard à l'importance des applications correspondantes (bases documentaires, banques de données).

2) conception descendante.

2.1) descente qualitative:

L'idée de base a été de considérer le problème de la recherche d'informations dans un texte comme un problème d'ANALYSE SYNTAXIQUE ET SEMANTIQUE d'un langage. La solution proposée est dès le départ très PROGRAMMABLE. Une requête est transformée en une grammaire, elle-même représentée sous la forme d'un ensemble d'instructions d'une machine spécialisée, dont la puissance minimale est la reconnaissance d'un langage d'expressions régulières.

Dès cet instant, les avantages de cette solution sur les autres -généralement beaucoup plus CABLEES - sont décisifs, en particulier le fait que le temps de réponse a une borne indépendante de la complexité de la question et permettra éventuellement un travail AU VOL sur des données séquentielles.

2.2)- descente quantitative statique:

Il s'est agi d'optimiser le cout de la représentation des automates de reconnaissance en place mémoire.

Pour cela, des mesures ont été effectuées, montrant en particulier l'importance d'une différentiation des représentations des parties arborescentes et des parties linéaires des automates.

2.3)- descente quantitative dynamique:

Ce point interfère avec le précédent. Il s'est agi

d'amèliorer la vitesse de l'algorithme d'analyse.
Pour cela, deux types de reprèsentations ont ètè retenues: les automates dichotomiques et les automates tabulès.

De plus, la notion de PREFILTRAGE des donnèes a ètè proposèe. Elle permet de diminuer trés simplement et d'une maniére considérable les contraintes de vitesse pour l'analyse gènèrale, et s'applique dans de nombreux cas en pratique.

Le rèsultat de toutes les phases prècèdentes aboutit à la notion de FILTRE, opèrateur matèriel capable d'effectuer AU VOL la quasi-totalitè des opèrations de lecture de fichiers, capable de compromis prix-performance trés satisfaisants, et supèrieur aux autres solutions proposèes auparavant dans la littèrature.

3) Conception ascendante.

Une fois la notion de filtre obtenue par l'analyse descendante, la dèmarche suivante est d'essayer d'en tirer le meilleur profit, et de tenter de faire profiter de ses performances et de sa simplicitè d'autres domaines que ceux pour lesquels il ètè conçu initialement.

3.1) conception ascendante logicielle

Sue le plan logiciel, outre l'écriture d'un langage de base pour traduire des requètes dans un langage de filtrage de fichiers sèquentiels, nous avons montrè l'intèrèt du filtrage pour gèrer les fichiers à accès aléatoire, et en particulier nous avons proposè une solution originale de gestion complète (insertion, suppression, interrogation) de fichiers multicritères trés performante qui utilise des structures de donnèes et des algorithmes simples (les RESUMES) et particuliérement bien adaptès aux filtres.

3.2) conception ascendante matèrielle

Le filtrage ètant à la base une opèration UNAIRE sur un

fichier, nous avons étudié comment réaliser -en réutilisant un filtre -, l'opération binaire fondamentale de l'algèbre relationnelle, la JONCTION.

Nous avons vu qu'il suffit de l'effectuer en DEUX filtrages, à condition d'utiliser le résultat du premier comme critère de filtrage du second. Des représentations des automates adaptées à ce mode de travail ont été évaluées, et la faisabilité de cette méthode a été montrée. En sous-produit, une solution pour réaliser la PROJECTION a été obtenue.

En conclusion, le filtrage apparait comme un mécanisme assez universel pour gérer des systèmes de bases de données, et le filtre est un OPERATEUR très utile dans toute architecture devant traiter des ensembles de données.

Le filtre est un peu aux traitements non numériques ce que les unités arithmétiques et logiques ou l'addition-décalage sont aux traitements numériques.

REFERENCES

REFERENCES

[20] SUCHARD,VIDALIN,QUANG Brevets français BF 74 39592 et BF 75 33423

[21] HSIAO The architecture of a database computer. Computer and information science center. Ohio state University. Sept 76

[22] DEFIORE,BERRA A data management system utilizing an associative memory. AFIPS Vol 42 1973

[23] PROPAL . Manuel de Rèfèrence. CIMSA Velizy France

[24] COPELAND,LIPOVSKI,SU The architecture of CASSM. 1st Symposium on computer architecture Dec 73

[25] OZKARAHN,SCHUSTER,SMITH Rap, An associative processor for Database Management. AFIPS Vol 44 1976

[26] ROBERTS A specialized computer architecture for text retrieval. 4th workshop on computer architecture for non-numeric processing. Aout 1978.

[27] HSIAO Editor. Database machines are coming. Computer Vol 12, n° 3. Mars 79

[28] KELLER Etude statistique de textes administratifs. Rapport de fin d'études. ENSIMAG Grenoble Juin 70

[29] HASKINS Hardware for searching very large text databases. 5th workshop on computer architecture for non-numeric processing. Asilomar Mars 1980

[30] Rapport de fin de contrat Laboria/CSEE sur l'étude d'une machine pour le renseignement téléphonique. IRIA Decembre 76.

[31] ROHMER,TUSERA Structures matérielles pour le filtrage d'informations textuelles. Note SCD n° 3. IRIA Decembre 76.

[32] KNUTH Searching and sorting. Addison Wesley 74

[33] LITWIN Hachage virtuel, une nouvelle technique d'adressage des mémoires. Thèse d'état. Mars 79. Université PARIS 6.

[34] BRUSQ,POIGNET le terminal de Télétexte ANTIOPE. Revue de radiodiffusion-télévision n° 49 1977

[35] YAO The database machine DIALOG. 5th workshop on computer architecture for non-numeric processing. Asilomar 1980.

[36] TUSERA Le Filtre de TREFLE. Note SCD IRIA Juin 77

[37 38 39] EL MASRI Conception, Simulation et Intégration Du Tampon. Notes SCD 6,7,8 IRIA

[40] TUSERA,HYAFIL Brevet Français 79 15753

[42] GALLAIRE Communication Orale Octobre 78

[43] GAUDEUL, ROHMER Description du module de calcul de la machine TREFLE Note interne INRIA 1980

[45] ANCEAU Communication Orale Decembre 79

[46] BANCILHON,SCHOLL A database machine architecture: VERSO. SIGMOD conference. Santa-Monica 1980

[47] MC CARTY et al. LISP 1.5 Programming manual MIT Press

Cambridge 1962

[48] SCHWAAB Contribution à l'étude d'un processeur de consultation de données. Thèse de 3ème cycle. Univ de Nancy. Juin 1979

[49] COMTE,HIFDI,SYRE The Data Driven LAU multiprocessor IFIP 80

[50] Voir référence 72

[51] KNUTH et al. Fast pattern matching in strings SIAM J. Comput, Vol 6, N° 2 Juin 1972

[52] Product Specification 9766 storage module drive. Control Data Corporation 1975

[53] PEANO Sur une courbe qui remplit toute une aire plane. Math. Annalen 36. Traduit dans 'Selected Works of Peano', Toronto University Press. 1972

[54] QUINQUETON Un algorithme adaptatif de balayage de Peano. Rapport LADORIA 344 Fevrier 79 IRIA

[55] VEILLON Utilisation de l'analyse linguistique automatique dans les systèmes documentaires. Atelier SESORI Saint Vallier Juin 78

[56] VIAULT La machine SARI Journées AFCET sur les machines bases de données. Mai 1979 Paris

[57] ROHMER The APL2M system. APL congress Rochester juin 79

[58] BOUTRY Manuel de référence APL2M . GIXI Orsay France 1980

[59] TUSERA Example of transformation of a derivation tree

for an expression by semantic attributes. IFIP 74.

[60] FORTIN, TUSERA The APL COMPILE operator. Note interne IRIA 1979

[69] TUSERA Description d'une méthode d'interprétation de APL par les attributs sémantiques. Thèse de docteur-ingénieur. Université Paris 6. juin 1974

[70] TUSERA Experimental comparison of two interpretation methods for APL. APL congress 75 Pise.

[71] MARCHAL Data structures techniques in an implementation of APL. Univ of Utah. Juin 73

[72] KAHN,MC QUEEN Coroutines and Networks of Parallel Processes. IFIP 1977

[73] IVERSON Operators and functions. APL congress Rochester Juin 79.

[74] DELOBEL Contribution théorique à la conception et à l'évaluation d'un système d'informations appliqué à la gestion. Thèse d'état. Université de Grenoble 1973.

[75] BANCILHON on the completeness of a relationnal query language. Rapport Laboria . Juin 1977 IRIA

[78] LE DISQUE OPTIQUE Revue technique Thompson 1978

[79] GUEUGNON,FAUGERAS Liaisons par fibres optiques appliquées à l'informatique. Congrès AFCET 80 Nancy

[87] IVERSON A programming Language. Wiley and Sons. New York 1972

[88] CODD A relationnal model of data for large shared data banks. CACM Vol 13 n° 6. 1970

[89] ROHMER,TUSERA Brevet français 79 15752

[90] PAIR Communication Orale Juin 1980

[91] MILNER An analysis of rotating storage acces scheduling in a multiprogrammed information retrieval system. Report UIUCDCSR-76-826 Univ of Illinois Sept 1976

[92] RUGGIU Description sémantique des fonctions primitives de APL. Revue technique Thomson CSF. Vol 4 n° 1 mars 1972

[93] JARAY Le langage SNOBOL 4, ses applications, son implémentation. Thèse de 3ème cycle. Univ de Nancy juin 75

[94] GRISWOLD et Al The Snobol 4 Programming language . Prentice Hall 1971

ANNEXES

ANNEXE A1 : PROGRAMME DE CREATION D'UN AUTOMATE

TABULE

Les deux tableaux AT et CDC ont le sens défini en 3 4.

NEXTIAT pointe sur le mot suivant libre dans AT et NEXTICDC fait de même pour CDC

Les états 1 à 6 de l'algorithme correspondent aux étiquettes R1 à R6 dans le programme AT1

RIAT et RICDC désignent la position courante des pointeurs sur AT et CDC lors du parcours de l'automate.

RESUL contient la valeur de l'état courant (de 1 à 6)

La fonction X SETTYCDC CONTCDC (resp CONTAT) indique que le mot X de CDC pointe sur CDC (resp AT)

La fonction X SETTYAT Y fait de même pour le mot X de AT

Les adresses pointant vers CDC sont représentées par leur opposé, celles pointant vers AT par leur valeur absolue.

Les fonctions KH sont des fonctions de comptage.

Les caractères sont décomposés en 4 tranches de 2 bits

N représente la valeur au dessous de laquelle les caractères ne sont pas découpés.

Le programme principal est PLACE.

Le corps de l'algorithme est dans AT1

La fonction RANGEAT C range le caractère C dans AT à partir de la valeur courante de BASEAT.

La fonction I NOUVELAT CARAC crée I nouveaux noeuds de 4 adresses dans AT pour y ranger les 2×I derniers bits du caractère C.

```
      ∇PLACE[⎕]∇
    ∇ PLACE;NOM
[1]   NEXTIAT←NEXTICDC
[2]  DEB:NOM←(' EANSIRTLOUDCPMVQGFBHXJYKZW/'\NXTNOM),0
[3]   CAM←CCAR
[4]   →(27≤NOM)/0
[5]   (NEXTIAT+NEXTICDC)←CCAR
[6]   RESUL←2
[7]   RIAT←0
[8]  AGAIN:
[9]   →(0=⍴,NOM)/DEB
[10]  CCAR←CCAR+1
[11]  CAR←NOM[0]
[12]  HISTO[CAR]←HISTO[CAR]+1
[13]  NOM←1↓NOM
[14]  AT1
[15]  →AGAIN
    ∇
```

```
[1]    →(R1,R2,R3,R4,R5,R6)[⍳1+RESUL]
[2]   R1:KH 1
[3]    AT[RIAT]←NEXTICDC
[4]    RIAT SETTYAT CONTCDC
[5]    CDC[NEXTICDC]←CAR
[6]    NEXTICDC SETTYCDC CONTCDC
[7]    RICDC←NEXTICDC
[8]    NEXTICDC←NEXTICDC+1
[9]    RESUL←4
[10]   →SUITE
[11]  R2:KH 2
[12]   BASEAT←⍳AT[RIAT]
[13]   RESUL←RANGEAT CAR
[14]   →SUITE
[15]  R3:KH 3
[16]   →(~CAR=⍳CDC[⍳AT[RIAT]])/R31
[17]   →(~AT[RIAT] EQTYCDC CONTAT)/R311
[18]   KH 4
[19]   RESUL←5
[20]   RICDC←⍳AT[RIAT]
[21]   →SUITE
[22]  R311:KH 5
[23]   RESUL←6
[24]   RICDC←⍳AT[RIAT]
[25]   →SUITE
[26]  R31:KH 6
[27]  ⍝LES 2 CARAC ≠,ARBORISER
[28]   AT[RIAT]←4 NOUVELAT⍳CDC[W←⍳AT[RIAT]]
[29]   RIAT SETTYAT CONTAT
[30]   →(~W EQTYCDC CONTCDC)/R32
[31]   KH 7
[32]   AT[NRIAT]←W+1
[33]   NRIAT SETTYAT CONTCDC
[34]   →SUITER3
[35]  R32:KH 8
[36]   AT[NRIAT]←⍳CDC[W+1]
[37]   NRIAT SETTYAT CONTAT
[38]  SUITER3:KH 9
[39]   BASEAT←⍳AT[RIAT]
[40]   RESUL←RANGEAT CAR
[41]   →SUITE
```

```
[42] R4:KH 10
[43]  CDC[RICDC+1]←CAR
[44]  RESUL←4
[45]  RICDC←RICDC+1
[46]  NEXTICDC←RICDC+1
[47]  RICDC SETTYCDC CONTCDC
[48]  →SUITE
[49] R5:KH 11
[50]  BASEAT←⌈CDC[RICDC+1]
[51]  RESUL←RANGEAT CAR
[52]  →SUITE
[53] R6:KH 12
[54]  →(~CAR=⌈CDC[RICDC+1])/R61
[55]  RESUL←5+CDC[RICDC+1]×0
[56]  RICDC←RICDC+1
[57]  →SUITE
[58] R61:KH 13
[59]  RICDC SETTYCDC CONTAT
[60]  W←4 NOUVELAT⌈CDC[RICDC+1]
[61]  →(~(RICDC+1) EQTYCDC CONTCDC)/R611
[62]  KH 14
[63]  ATENRIAT]←RICDC+2
[64]  NRIAT SETTYAT CONTCDC
[65]  →SUITER6
[66] R611:KH 15
[67]  →(0=CDC[RICDC+2])/R6111
[68]  KH 16
[69]  ATENRIAT]←⌈CDC[RICDC+2]
[70] R6111:KH 17
[71]  NRIAT SETTYAT CONTAT
[72] SUITER6:KH 18
[73]  CDC[RICDC+1]←W
[74]  (RICDC) SETTYCDC CONTAT
[75]  BASEAT←W
[76]  RESUL←RANGEAT CAR
[77]  →SUITE
[78] SUITE:
     ∇
```

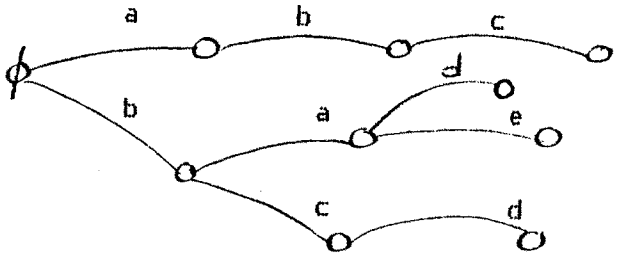
```
    ∇ R←RANGEAT CARAC;TR;W;Z;WW;I
[1]   BASEAT←⌊BASEAT
[2]   →(~CARAC<N)/AL1
[3]   KH 28
[4]   →(~AT[BASEAT+CARAC]≠0)/AL2
[5]   RIAT←BASEAT+CARAC
[6]   R←(2 3)[RIAT EQTYAT CONTCDC]
[7]   →0
[8]  AL2:
[9]   RIAT←BASEAT+CARAC
[10]  R←1
[11]  →0
[12] AL1:
[13]  BASEAT←BASEAT+N
[14]  TR← 4 4 4 4 ⍴CARAC
[15]  →(M1,L1)[0≠Z←AT[W←BASEAT+TR[0]]]
[16] L1:
[17]  →(M2,L2)[0≠Z←AT[W←Z+TR[1]]]
[18] L2:
[19]  →(M3,L3)[0≠Z←AT[W←Z+TR[2]]]
[20] L3:
[21]  →(M4,L4)[0≠Z←AT[W←Z+TR[3]]]
[22] L4:KH 23
[23]  RIAT←W
[24]  R←(2,3)[RIAT EQTYAT CONTCDC]
[25]  →0
[26] M1:I←1
[27]  →S1
[28] M2:
[29]  I←2
[30]  →S1
[31] M3:
[32]  I←3
[33]  →S1
[34] M4:RIAT←W
[35]  KH 24
[36]  R←1
[37]  RIAT SETTYAT CONTCDC
[38]  →0
[39] S1:KH 24+I
[40]  WW←(4-I) NOUVELAT CARAC
[41]  AT[W]←WW
[42] S2:
[43]  R←1
[44]  RIAT←NRIAT
[45]  RIAT SETTYAT CONTCDC
[46]  →0
```

```
      ∇NOUVELAT[⎕]∇
    ∇ R←I NOUVELAT CARAC;W
[1]   R←NEXTIAT
[2]   →(~I=4)/AL1
[3]   →(~CARAC≥N)/AL2
[4]   NEXTIAT←NEXTIAT+N
[5]   →AL1
[6]  AL2:
[7]   NRIAT←NEXTIAT+CARAC
[8]   NEXTIAT←NEXTIAT+4+N
[9]   →0
[10] AL1:
[11]  KH 10+I
[12]  AT[W←NEXTIAT+((-I)↑(4 4 4 4 ↑CARAC))+4×⍳I]←(NEXTIAT+
      4+4×⍳I)
[13]  NEXTIAT←NEXTIAT+4×I
[14]  NRIAT←(NEXTIAT-4)+4↑CARAC
    ∇
```

ANNEXE A2 : ALGORITHME DE KNUTH POUR LE CALCUL DES REGRESSIONS.

Etant donnè un ensemble de mots, l'automate fini qui permet de reconnaitre si une chaine est EGALE à l'un de ces mots est un arbre. Exemple:

Automate reconnaissant les mots (abc,bcd,bae,bad):



Pour chercher si la chaine analysèe CONTIENT un de ces mots, il faut complèter l'automate par des retours arrières en cas d'èchec

Dèfinitions:

PERE(X) dèsigne l'ètat père de l'ètat X

SUC(X,a) est le successeur de l'ètat X lorsque le caractère en entrèe vaut a. (ce successeur peut être vide)

EI est l'ètat initial

Pour chaque ètat X, il faut calculer son retour arrière RA[X].

Voici l'algorithme récursif qui définit RA(X).

Les retours arrières sont calculés par niveaux successifs à partir de EI. (le niveau i comportant tous les états à une profondeur i).

Niveau 0 : RA(EI) = EI

Niveau 1 : Si PERE(X) = EI Alors RA(X) = EI

Niveau i : Si l'on connait les RA de tous les niveaux < i, alors le retour arrière d'un état X du niveau i est donné par l'algorithme suivant:

Soit a le caractère (unique) tel que SUC(PERE(X),a) = X

Z ← RA(PERE(X))

ENCORE:

Y ← SUC (Z,a)
Si Y n'est pas vide alors RA(X) ← Y
Sinon Z← RA(Z)
Si Z = EI, alors RA(X) ← EI
Sinon aller à ENCORE

dernière page de la thèse

AUTORISATION DE SOUTENANCE

VU les dispositions de l'article 5 de l'arrêté du 16 Avril 1974,

VU les rapports de Messieurs :

- F. ANCEAU, Professeur à l'Institut National Polytechnique de GRENOBLE
- Cl. DELOBEL, Professeur à l'Université Scientifique et Médicale de GRENOBLE
- C. PAIR, Professeur à l'Institut National Polytechnique de Lorraine

Monsieur Jean R O H M E R

est autorisé à présenter une thèse en soutenance pour l'obtention du grade de DOCTEUR D'ETAT, discipline SCIENCES.

Grenoble, le 3 Décembre 1980

Le Président de l'U.S.M.G.

Le Président de l'I.N.P.G.

Ph. TRAYNARD
Président
de l'Institut National Polytechnique