Category theory for (big) data modeling and model's transformation

Heng Zhao



UNIVERSITE DE HAUTE ALSACE

ECOLE DOCTORALE: MATHÉMATIQUES, SCIENCES DE L'INFORMATION ET DE L'INGÉNIEUR

INSTITUT DE RECHERCHE EN INFORMATIQUE, MATH´EMATIQUE, AUTOMATIQUE ET SIGNAL

THESE PRESENTEE POUR OBTENIR LE GRADE DE

Docteur de l'Universite de Haute Alsace

Discipline: Informatique

Category theory for (big) data modeling and models' transformation

Heng ZHAO

SOUTENUE PUBLIQUEMENT LE 05/07/2019

Membres du jury:

Mr. Horatiu Cirstea, Professeur, rapporteur, Université de Nancy
Mr. Amir HAJJAM EL HASSANI, Maitre de conférences, rapporteur, UTBM
Mr. Pierre-Alain Muller, Professeur, examinateur, UHA
Mr. Michel Hassenforder Professeur, Directeur de thèse, UHA

Acknowledgment

As a thesis which represents in Category theory for big data modeling and models' transformation, this thesis was finished after 3 years of work at Institut IRIMAS, Université de Haute Alsace.

I am particularly honored by the presence of this jury of professors:

Pierre-Alain Muller, Professor of université de Haute-Alsace,

Horatiu Cirstea, Professeur of Université de Nancy,

Amir HAJJAM EL HASSANI, Associate professeur of UTBM,

I would like to thank Mr. Michel Hassenforder for his concern and help without which I would not have been able to to complete the research work on this subject.

I am also very grateful to the China scholarship council for my three years of financial support.

In the end, I want to thank my parents. They have given me the spiritual support to complete this subject. In addition, I would also like to thank all the friends who have cared and helped me during my PhD.

Contents

List of Figures

1. Introduction

- Thesis introduction
- Motivation
- Summary

Chapter 1

Introduction

Big data is a collection of data that cannot be captured, managed, and processed with regular software tools over a period of time. Big data requires a new processing model to have a more efficient capability to process large-scale data. Hence, the data model is the main line of big data technology development, the core and foundation of large-scale databases. More specifically data structure and data manipulation are the two basic elements of the data model. The data structure describes the object type of the static characteristics of the data and constitutes the basic component of the database. The data manipulation, which is also named ”query” in the report, is the dynamic characteristic of the database. It defines the set of operations that the database can perform, clarifies the exact meaning of the data operations, the operational symbols, and the rules of the operations. Our work is concentrated on the large amounts of data that directly impact the performances of the programs (e.g. to query a specific information) and which require specific architecture to improve them, e.g. use of graph databases or maybe use of distributed concurrent computations. Though a lot of technologies are available today to put BigData into practice, usable theories to better understand the benefits and the limitations of each architecture, to identify possible improvements or means to combine them are more rare. Our research presents the capabilities offered by category theory together with a functional programming language (to implement the concepts and facilitate experimentation) to solve these limitations. In particular, it ex-

plains how functors can change data structures (e.g. various representations of set) and how natural transformations can be used to change data structures or shift programs applicable to a particular data structure to another program for an other data structure. The concept of natural isomorphism is then established to prove that two data structures represent the same information, or that two programs are equivalent. The equations representing programs can serve to calculate computation time and to compare the performances of two programs, then show that a natural transformation can be an optimization. An advantage of Category Theory is to be easily and safely translated in most of the functional programming languages, what is interesting to make experiments and proposes new architectures or tools to Big Data community. As an illustration the thesis proposes an optimized (by way of natural transformations) implementation of an information server and its query language in the Haskell functional programming language. The other interests of our research are then to detail the implementation and to give a comparison of our approach and of standard tools available in the market. As a complement, it shows that these programs can implement complex algorithms (such as unification) by using the capabilities brought by the concepts (e.g. functors and higher-order functions). The comparison step finally shows that: 1) The program presented is able to deal with a large set of data. 2) Some transformation are better. 3) Surprisingly, the comparison of the time execution obtained by five well-known data processing tools (Sqlite Studio, MonetDB, MongoDB, Neo4J, Cassandra) is favors functional programming.

The document is divided into three parts. Chapter 2 starts by introducing the contribution of the thesis in context with ”Related works” and introduces models commonly used by information systems and takes a query by the standard tools to extract the time complexity. Then, it presents the fundamental concepts of category theory and how they can be implemented in a functional programming language. In particular, this chapter details how ”functors” can transform data structures, and how ”natural transformations” change a program using a data structure to a new more efficient one using another structure. Chapter 3

explains how Category Theory can be used to define an efficient information server and its query language (based on unification). Chapter 4 presents the dataset considered in the experiments and gives a comparison of the performances (time to answer a query) obtained with the system proposed and standard tools (Sqlite, Mongodb, Neo4j, MonetDB,Cassandra). In this chapter, it is also entitled "Discussion" and examines in detail the benefit/limitation of the elements proposed, and shows how to apply them in other contexts. Finally, a conclusion summarizes the main elements presented: a theoretical approach of Big Data ; i.e. an efficient information system with a comparative study, and describes some of the perspectives considered.

2. State of art

- Related works
- Database for Big Data
- Category theory and Haskell
- Data and Querying Modeling

Chapter 2

State of art

2.1 Related works

Big Data is centered on very large datasets and a sample illustration is presented in the Figure 2.1. The query extracts the first entry from a subset of more than 900 million entries. These entries can be entity-centric knowledge resources, including entities of a particular thing, relationships of physical logical connections, or classifications used to semantically mark entities. Therefore, in the field of digital engineering, the collection of data is usually described as the form of entities, relationships, classifications and other elements. However, as Hal Varian (Google's chief economist) said [68]: although Big Data is widely used, what's lacking is the ability to obtain information from it. The data search engine "Google", which is shown in Figure 2.1 is the reorganization of data in the form of a network. Each piece of data in the network represents an entity, and the relationship between the data represents the relationship between the entities. As it is shown in Figure 2.1, building a data set quickly with a specific keyword (eg., Category theory) involves many more resources. In general, as explained in [58], dealing with a huge amount of data requires specific architecture both for hardware (e.g. cloud computers by using Map Reduce [59]) and for software (e.g. graph database servers). Though many theoretical models are then proposed to get a better value from all the data available, theories able to formalize the concepts under the tools

commonly used to manage or query the data are more rare. The aim of this thesis is to explain how Category Theory can solve such a limitation and, associated to a functional programming language, be used for instance to propose efficient information servers for large datasets (e.g. reducing the 0.42s in the Figure 2.1) or to shift data between various formats this, by combining natural transformations.

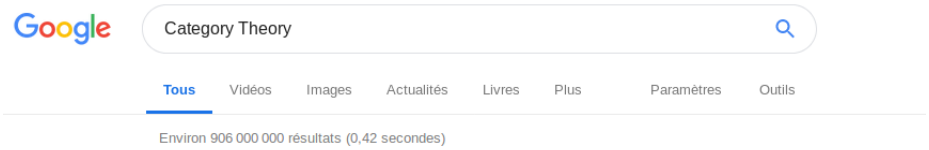


Figure 2.1: Sample performance

Of course, the use of Category Theory for software development is not new. In particular, this theory has already shown its advantages in the domain of "program calculation" with for instance [60] or [61], in the domain of Model-Driven Engineering [62], etc. The concepts of the theory has also been implemented in some programming languages, such as ML or Haskell, and can be used directly in these languages, e.g. [63]. At an extreme, the concepts have themselves been used to define a specific programming language in [64]. Category Theory has also lead to specific platforms for the management of (graph) data models [65] and query [66]. The contribution of this thesis is to go further considering big datasets - which have not been considered in the above works, and by making possible the interchange of data with more classical tools found in the Big Data community.

2.2 Databases for Big Data

2.2.1 Introduction

Since the 1970s, several database structures have been proposed. The first one is Hierarchical Database, which arranges data in a tree structure. This tree structure is similar to folders and files on a computer. Hierarchically arranged

data is often described as having only parent/child relationships. Later, network database was an evolution of database model where multiple data can be linked by themselves. The model can be viewed as an upside-down tree where each member information is the branch linked to the owner, which is the bottom of the tree. In the year of 1970s, the relational database was proposed which was based on an algebra and a universal query language was designed. Relational database (SQL): E.g. MySQL, Oracle, SQLServer, SQlite (and a lot of other) uses SQL language to achieve the querying language. However, with the development of Web 2.0 and some new structures of database and "NoSQL" have appeared. On the other hand, NoSQL database have now four types: Key-value databases, Document oriented databases, Graph databases, Column stores databases. Both of them are depend upon the JSON format. Each type of database has their own properties: ACID or BASE.

With the web 2.0, the speed of execution time and the partition tolerance has become more and more important. Hence, a new property CAP is coming for the recent database.

SQL and NOSQL

SQL is the abbreviation for Structured Query Language [1]. It could be used for organizing, managing, storing and retrieving data stored by a computer database. As the name implies, SQL is a computer language which can be used to interact with a database. It is a special purpose programming language. It does not require the user to specify the data storage method nor necessarily to understand the specific data storage methods [4]. Therefore, different database systems with completely different underlying structures can use the same structured query language. Structured query language statements can be nested, which gives it great flexibility.

NoSQL means "not only SQL"[4]. It provides a mechanism for storage and retrieval of data that is different than the tabular relations. With the rise of the internet web2.0, the traditional databases meet a problem: The capability to deal with very large datasets. The non-relational database due to its own

characteristics have been developed very fast. The aim of NoSQL databases is to solve the challenges of multi-data types in large scale data collections, especially the problems for big data application.

ACID property

ACID property is mainly implemented by a relational database management system (RDBMS). ACID [31] is Atomicity, Consistency, Isolation, Durability. The meaning of the four rules are:

Atomicity

All operations in the entire transaction are either completed in their entirety or are not completed at all. It cannot be stuck in the middle of a transaction.

Consistency

Transactions must always keep the system in a consistent state, regardless of the number of concurrent transactions at any given time.

Isolation

If there are two transactions running at the same time and performing the same function, the isolation of the transaction will ensure that only one transaction considers is using the system.

Durability

When the transaction is completed, all of the changes made by the transaction to the database are remained in the database.

The database management system uses logs to ensure the atomicity, consistency, and durability of transactions. The log records the updates made by the transaction to the database. If an error occurs during the execution of a transaction, it can be based on the log to undo the updates that the transaction of database and return the database to the initial state.

A Database Management System in Figure 2.2 has a database that stores information [1].

More precisely, the main components are:

- **Server:** A program (software) for receiving and processing requests made by other programs, or a device (computer).

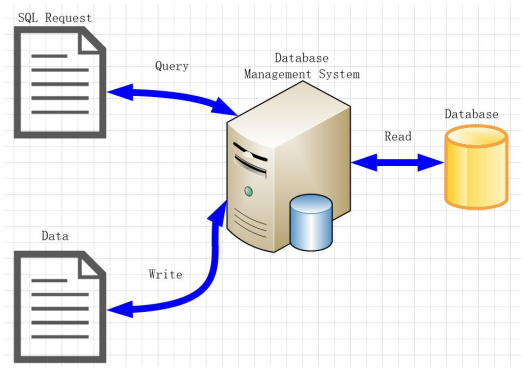


Figure 2.2: The database management system

- **Clients:** The program (software) that makes the request to the server, or the device (computer).

Distributed databases

If the user has only used one database to take the operation of read and write, the database could be overwhelmed. Most web sites have used master-slave replication technology to achieve the operation.

A better management system based on two main databases to conserve database in Figure 2.3 which could be used to improve performance. And then, it has used some attached databases to receive the data from Main database. Therefore, big data manipulation will use the database master-slave mode. It is relatively simple to expand the searching scope of the database. However, there is no easy way to solve the problem of big data writing.

In order to scale the data, the user could increase the main database from one to two. This could reduce the load on each main database by half. However, the update process will conflict, the data may cause inconsistencies. In order to avoid such problems, the requirement for each table is assigned to the appropriate request to the main database.

Splitting the database, respectively, on a different database server. This is shown in Figure 2.4 and Figure 2.5.

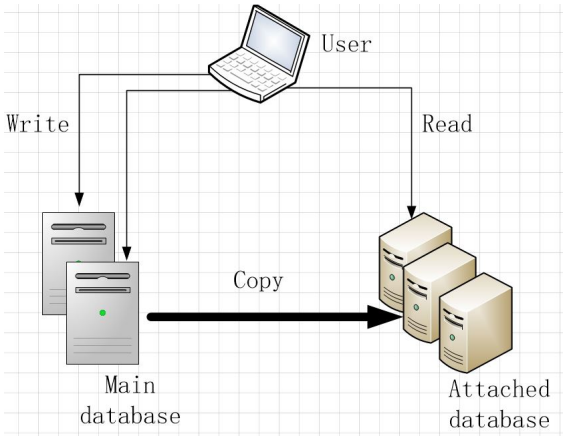


Figure 2.3: Main database management system 1

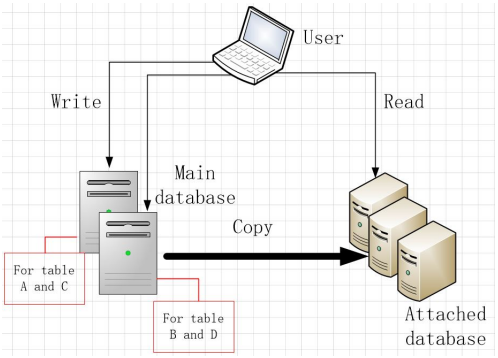


Figure 2.4: Main database management system 2

For example, tables are hosted only once and on a different database server. Database partitioning can reduce the amount of data on each server. This could be used to reduce input and output processing and achieve high-speed memory processing. However, if joining the results from the servers is needed the cost (time and space) could increase a lot. Relational databases [3] are widely used and could perform complex queries such as transaction processing and table joins. There is an architecture network in Figure 2.6 to explain the approach to query the databases in a different server.

When the user needs to query some information, they will use the different IP address to search the target databases.

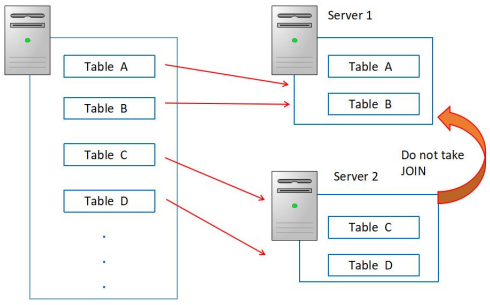


Figure 2.5: Database segmentation

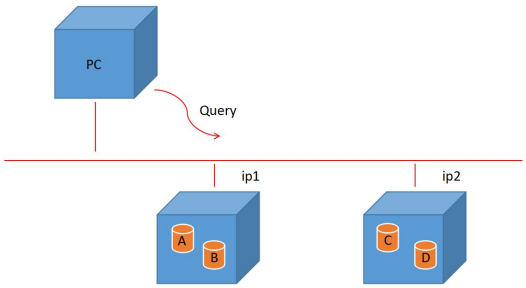


Figure 2.6: The approach to query the information in different servers

BASE property

The BASE model is logically the opposite of the concept of ACID model. BASE [1] stands for Basically Available, Soft-state, and Eventually Consistent. It sacrifices high consistency and obtains availability and partition tolerance. There may be momentary inconsistencies in the system's processing of requests. Each step in the system can record each temporary state. In the event of a system failure, unfinished requests can be processed from these intermediate states or returned to the original state, and finally obtain a consistent state.

CAP property

CAP property is the cornerstone of the NOSQL database. NoSQL is mainly a distributed system and the biggest difficulty is the synchronisation of each node. CAP [32] is an answer based on three elements: Consistency, availability

[1] https://www.abis.be/resources/presentations/gsebedb220140612nosql.pdf

and partition tolerance.

Consistency

For distributed storage systems, one piece of data tends to exist in multiple copies. In simple terms, consistency will allow the customer to modify the data (add/delete/change), or succeed in all data replicas either all failed. That is, the modification operation is an atomic operation for all copies of a piece of data (the entire system). If a storage system can guarantee consistency, then the data read and written by the customer can be guaranteed to be up-to-date. Two different clients can not read different copies from different storage nodes.

Availability

It means that when the client wants to access data, they can get a response. However, the availability of the system does not mean that the data provided by all nodes of the storage system are consistent. In this case, the system is available.

Partition Tolerance

If the storage system only runs on one node, either the entire system crashes or all goes well. The storage system for the same service is distributed to multiple nodes, there is a possibility of partitioning the entire storage system. For example, a disconnected network between two storage nodes forms a partition. In general, it is normal for the same data to be placed in different nodes in order to improve service quality. Therefore, it is normal to form partitions between nodes.The CAP principle means that these three elements can only achieve two points at the same time.

There are some examples below:

CA meets data consistency and high availability but no scalability

APs perform well in terms of performance and scalability, but they sacrifice their data consistency. Data synchronization between nodes is not as fast, but it can preserve the ultimate consistency of the data.

The CP satisfies data consistency and zoning. However, when the number of nodes reaches a certain number, the performance (that is, availability) will drop rapidly, and the network overhead between the nodes. The data between

the nodes needs to be synchronized in real time.

2.2.2 Relational databases

In 1970s, the concept of RDBMS was proposed to deal with large databases. Nowadays, there are lots of RDBMS such as MYSQL, SQL SERVER ORACLE,etc. and we will use SQLITE to perform query and data extraction. As its structure is convenient (No overheads due to network processing).

Relational Data structure

Relational database is a database that complies with ACID's relational database management system; Figure 2.7 shows the structure of a table in this database. It could be queried by SQL language.

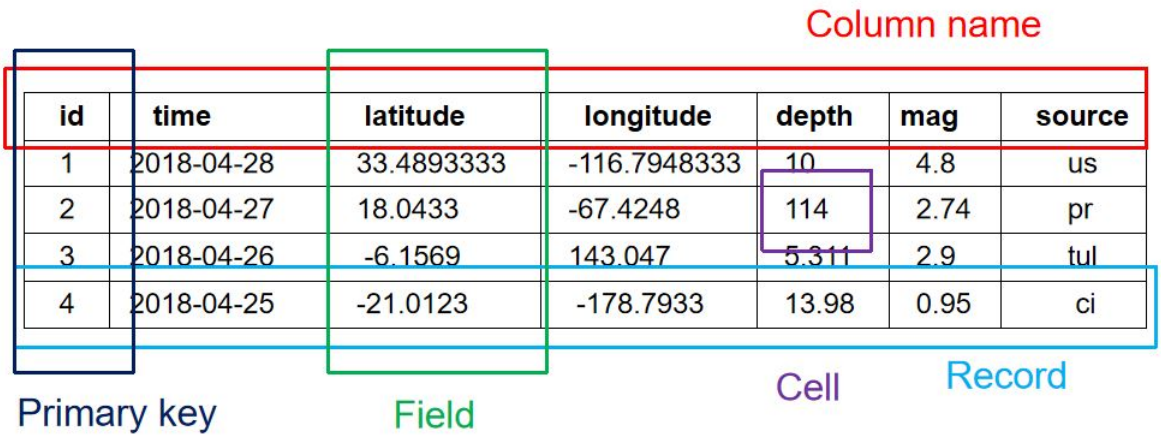


Figure 2.7: The structure of a table

There are some key words in the figure above.

- **Table:** A two-dimensional array composed of rows and columns.
- **Field:** Column of tables (vertical direction).
- **Record:** Row of tables (horizontal direction).
- **Cell:** The intersection of a column and a row.

- **Primary key:** This is a key in relational database to search one record. It is either an existing table column or a column that is specifically generated by the database according to a defined sequence.
- **Foreign key:** It is a column in a table, which points to the primary key in another table.
- **Schema:** A schema describes the set of tables, types for each column and the relationships between the different tables. An efficient schema requires the use of a structured approach and is validated by checking the normality level of the database.

Relational Algebra

Algebra is a basic mathematical branch with many research objects, such as number, quantity, algebra, relationship, equation theory, algebraic structure and so on. The study of algebra is not only on numbers, but also a variety of abstract structures. For example, an integer set is a collection of integer with addition, multiplication, and ordering is an algebraic structure. Algebra also exists in relational databases and it is called "Relational Algebra"[2]. It is a "Relation" $R \subseteq X \times Y$, which is a subset of the Cartesian product of X and Y. Set operations allow the results of multiple queries to be combined into a single result set. And set comprehension can be used by $\{f(x)|x \in r, p(x)\}$. Because there are operators (intersection, union, etc.) and operand (tables). This is a family of algebras with a well-founded semantics used for modeling the data stored in relational databases, and defining queries on it. There are some basic operations below:

Selection δ

Extract records satisfying a predicate. The set comprehension is: $\delta_p(r) = \{x|x \in r, p(x)\}$

Projection π

[2] https://cs.nyu.edu/courses/Fall12/CSCI-GA.2433-001/lecture4.pdf

Extract selected fields. The set comprehension is: $\pi_1(r) = \{\pi_1(x)|x \in r, p(x)\}$

Cartesian Product

$R \times S$ forms a (n + m) column tuple set. If R has K1 tuples and S has K2 tuples, $R \times S$ has K1 * K2 tuples. The set comprehension is: $r \times r' = \{(x, x)|x \in r\}$

Union

R, S have the same relational pattern (same elements, same structure), denoted $R \cup S$. Returns a collection of R or S tuples.

Intersection

R, S have the same relational pattern (same elements, same structure), denoted $R \cap S$. Keeps the common part of R and S

Set difference

R, S have the same relational schema (same elements, same structure), operation denoted by $R - S$, eliminates the part of R that belongs to S.

Figure 2.8 below show the six operations:

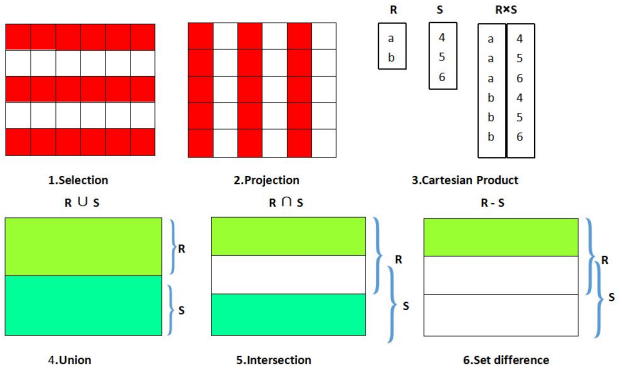


Figure 2.8: Basic operations

JOIN operations

A SQL join clause combines columns from one or more tables in a relational database. It creates a set that can be saved as a table. A JOIN is a means

for combining columns from one (self-join) or more tables by using common values, such as foreign keys and primary keys.

Inner Join

Inner join requires that the components to be compared in the two relationships must be the same attribute group. And remove duplicate attributes listed in the results. This is the same as the first table in Figure 2.9.

Semijoin

Semijoin returns the first table records. Table H returns only one record, even if several matching records are found in I. This is the same as the second table in Figure 2.9.

Outer joins

The joined table retains each row even if no other matching row exists. Outer joins subdivide further into left outer joins, right outer joins, and full outer joins, depending on which table's rows are retained (left, right, or both). The third table in Figure 2.9 shows the left Outer join. It could return all the values from an inner join plus all values in the left table that do not match the right table, including rows with NULL (empty) values in the link column. In contrary, the fourth table in Figure 2.9 shows the right Outer join which could return all the values from an inner join plus all values in the right table that do not match the left table.

The figure 2.9 show 4 different JOIN operations:

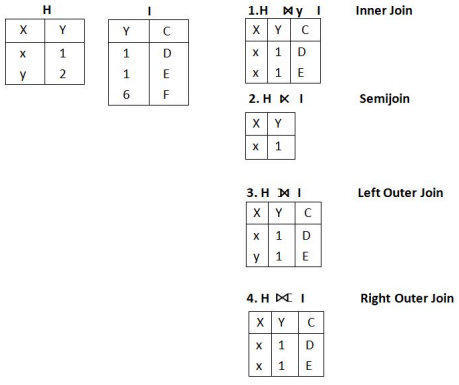


Figure 2.9: JOIN operations

Division

There is a table below to show Division operations:

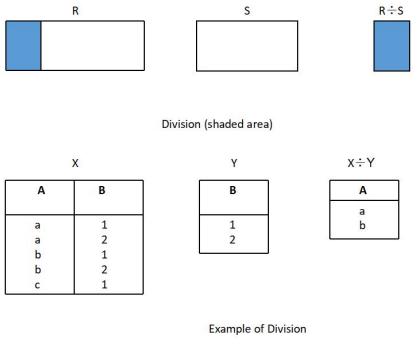


Figure 2.10: Division operation

As shown in Figure 2.10. Division operation means that we keep the same terms in X as in Y. Then store this in a table.

Foreign key

Foreign key in one table points to the Primary key in another table as illustrated in Figure 2.11.

Earthquake information:

id	Time	latitude	Longitude	depth	mag	Source
5a9d7c2ed6e7	2018-02-28	“33.4893333”	“-116.7948333”	“10”	“4.8”	“us”
5a9d7c2ed6e8	2018-02-27	“36.4092”	“-98.7856”	“8.3”	“0.9”	“ak”
5a9d7c2ed6e9	2018-02-26	“37.3671”	“-117.1218”	“32.47”	“2.5”	“us”

Orders:

id	Earthquake NO	Earthquake_id
69	67823	5a9d7c2ed6e8
21	93267	5a9d7c2ed6e7
177	12563	5a9d7c2ed6e9

Figure 2.11: Foreign key

The **”Id”** field in the ”Orders” collection points to the ”Earthquake_Id” field

in the ”Earthquake information” collection.

The **”Id”** field in the ”Earthquake information” collection is the Primary key in the ”Earthquake information” collection.

The **”Id”** column in the ”Orders” collection is the Foreign key in the ”Orders” collection.

FOREIGN KEY constraints can be used to prevent the destruction of connections between collections.

The **FOREIGN KEY** constraint also prevents illegal data insertion into the foreign key field, because it must be one of the values in the collection which it points to.

2.1.2.5 Example

Relational database uses SQL to interact and instructions are available to create the bases, tables and operations such as insert, remove, update data or query the database.

1) Create table:

```
CREATE TABLE earthquake (id INT PRIMARY KEY AUTO_INCREMENT,
                         time TIMESTAMP,
                         latitude REAL,
                         longitude REAL,
                         depth REAL,
                         mag REAL,
                         Source CHAR(2) )
```

In this example, the table is named ”earthquake” and each row has to describe uniquely an earthquake somewhere around the world. So additional typed information is required. First is id, an INT which will be our primary key and auto incremented when inserting values. Time is a date and a time when the earthquake occurs, and latitude, longitude and depth are REAL numbers for the geolocation. Mag represents magnitude and is a REAL. Last column,

the Source is just a country abbreviated by its two letters following the ISO 3166.

2) Insert values

```
INSERT INTO earthquake VALUES
 (2018-04-28, 33.4893333, -116.7948333, 10, 4.8, us)
```

3) Search value and this performance

```
Select * from earthquake where Source='us'
```

Corresponding to the set comprehension is:

$\{Allcolumn(x)|x \in y, columnSource(x) =' us'\}$.

Among them, $Allcolumn(x)$ means projection π and $columnSource(x)$ means selection δ.

We can use the code of SQL above to search for the useful information. For example, querying all of the earthquakes in USA on this database. The performance is $0.004s$.

2.2.3 Column-oriented databases

In these years, the development of data storage technology towards massive data, analytical data and intelligent data. As one of the type of NoSQL database, column store database stores a table in a sequence of columns. The main reason for its rapid development is its high efficiency of complex queries, fewer reading disks and less storage space. This part presents this type of technology and we used MonetDB to create and query again an earthquake information database.

Column concept

Column-oriented databases [16] [17] are databases that store data in column-related storage architectures. It is suitable for batch data query and processing.

Column store database store data as two-dimensional tables of columns. However, it is accessed as a one-dimensional string. Figure 2.12 shows an example of earthquake information.

A Column-store database stores the values in a column together and then stores the data in the next column. For example, the data in Figure 2.12. ”Time” column is 2018-04-28, 2018-04-27, 2018-04-26, 2018-04-25; ”Latitude” column is 33.4893333, 18.0433, -6.1569, -21.0123; ”Longitude” column is -116.7948333, -67.4248, 143.047, -178.7933; ”depth” column is: 10, 114, 5.311, 13.98; ”mag” column is: 4.8, 2.74, 2.9, 0.95; ”Source” column is us, pr, tul, ci; This is an intuitive version of Column-store database. On the other hand, Column database storage organizes and stores data according to the columns of database records. Each table in the database consists of a set of page chains. Each page chain corresponds to a storage column in the table, and each page in the page chain stores one or more values for this column (see in Figure 2.12).

time	latitude	longitude	depth	mag	Source
2018-04-28	33.4893333	-116.7948333	10	4.8	us
2018-04-27	18.0433	-67.4248	114	2.74	pr
2018-04-26	-6.1569	143.047	5.311	2.9	tul
2018-04-25	-21.0123	-178.7933	13.98	0.95	ci

Figure 2.12: An example of an earthquake information

In case of a query, only the columns involved in the query need to be accessed, this method of operation reduces the system’s I/O and optimizes the time to respond. This allows time for column databases to query data more quickly. Therefore, there are three advantages in using column store database:

1) Data storage and data compression: Each column is stored separately. Hence, the data in the same column has consistent data types, similar data characteristics and efficient compression. If the user squeezes a lot of data

together, more data could be stored / collected / used. Therefore, this approach of reading data can increase the speed of processing data. At the same time, the storage characteristics of the column database also facilitate the rapid query of the required columns. In addition, this type of storage only needs to read the relevant data and could write data from multiple entries.

2) Data management: Column store database is stored in the order of key value compression. It could give an index to all the columns. Hence, the data maintenance for this database is relatively simple.

3) Data querying: In this mode, In this mode, the query language just scans the page chain of database corresponding to the column. It does not need to access the full table. Thus, this method increases the speed of querying data.

MonetDB

MonetDB is an open source, column-oriented database management system. MonetDB was designed to provide high-performance query for big data, such as millions of rows and hundreds of columns of database. MonetDB is also a database management system that supports the SQL:2003 standard, despite its NOSQL vision and also XML, RDF and SPARQL[22].

Storage model[3] of MonetDB [16]is very different from traditional databases. It means that MonetDB table uses vertical storage, each column is stored in a table (surrogate, value) named BAT (Binary Association Table). The left column presents the proxy number or Object id. It is called the header; the right column is the tail. Figure 2.13 describes two different Column stores.

The first one is a Column store with Proxy Ids. The second table is a Column store with Object Ids. MonetDB performs low-level relational algebra called BAT algebra. During execution, data is often stored in (intermediate) BAT[18], even if the result of the query is also a collection of BAT. BAT storage uses two simple memory arrays. One array is the header and the other is the tail (column). MonetDB divide the array into two sections for different types of length. A segment with concatenated data values and another segment

[3]https://www.monetdb.org

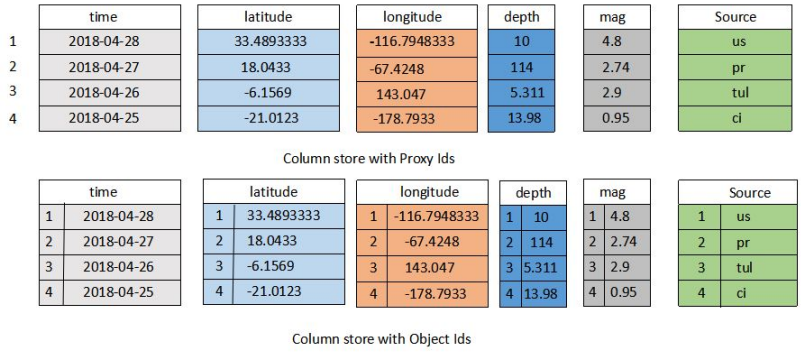


Figure 2.13: An example of two different Column store

with offsets into the former. For a small "string value" table, the combination implements a value as in a dictionary.

MonetDB uses a memory-mapped file to implement storage columns. The storage columns are optimized to the unique situation where the columns of the proxy are ascending numbers $(0, 1, 2...)$; in some cases the proxy columns can be kept ambiguous. The proxy column query becomes a way to read the tail with a fast array subscript. In fact, this mechanism of using arrays in virtual memory to use the MMU (memory management unit) to quickly map hardware addresses to disk addresses provides the database with an efficient database search mechanism. On the other hand, MonetDB is based entirely on memory-mapped files. Once you need to swap to disk, the performance is surprised. This means that the amount of data to be processed does not exceed the total amount of memory before using MonetDB. And then, a Column store database is suitable for the database which has many tuples or records on few attributes.

For example, if the query needs to search the third value in column of "latitude"[17]. The assumption here is the column is fixed-width. Hence, the result could be obtained by: $start(latitude) + 3 \times width(latitude)$ where $start(latitude)$ and $width(latitude)$ means the start address of the column "latitude" and the width of the column "latitude" respectively.

Join operation

There are two basic elements in MonetDB[21]. The table storage which is used in MonetDB is an associative array. The first element is Storage key (In MonetDB, it could be Proxy Id or Object id). Every data value with the same column has the same storage key. Data in different columns and in the same segment with matching storage keys belong to the same row. It is the same as the primary key in RDBMS.

The second one is Join Index. Join Index is used to reconstruct all of the records in a table from its various projections. For example, if there are two columns C1 and C2 which belong to a table C. A join index from M series in C1 to the N series in C2 is A collection of M tables. One value of each series of C1, H consist the form of row:

$(H : Object\ id\ in\ C2,\ s :\ Storage\ key\ in\ serie\ s)$

A new given tuple in a series of C1 contains the Object id or Proxy id of the corresponding (joining) tuple in C2. This is always a one-to-one mapping because all join indexes are located between the projections anchored on the same table.

Example

1) Create table

In this article, the encoding rule is respected to SQL language. Table is one of the basic elements of a Column store database. The code below is used to create a table named earthquake.

```
CREATE TABLE "earthquake" (
        time TIMESTAMP,
        latitude REAL, longitude REAL, depth REAL,
        mag REAL, Source CHAR(2)
    );
```

2) Values insertion

The code below is used to insert the value in the table ”earthquake”. The order has to be the same as in the table insertion.

```
INSERT INTO "earthquake" VALUES
(2018-04-28, 33.4893333, -116.7948333, 41.32, 4.9, "us"),
(2018-04-28, 38.8003349, -122.7588348, 87.02, 4.5, "us"),
(2018-04-29, 38.8003349, -122.7588348, 0.46,2.35, "hv");
```

3) Select operation

The operation by selecting value is also the ”Join” operation to a new table. For example, the code below is to select the earthquake which happened in ’2018-04-28’. The operational time is 22.151ms.

```
select * from earthquake where Source = "us";
```

4) Import CSV file

From this part, the sample has used the database with earthquake information by the United States Geological Survey. By using this database, we will verify the execution time to search the value in a big database.

The first step is to import this file into MonetDB. As before, creating the table is done by using all of the different series names as necessary. The name of this table is usgs. The code is below:

```
CREATE TABLE usgs (
      time varchar(50),
      latitude char(15),
      longitude numeric(10,7),
      depth char(15),
      mag char(15),
      magType string,
      id varchar(50),
      type string,
      Source string
);
```

In this code: - VARCHAR ’(’ length ’)’ is UTF-8 character string with length upperbound limit.

- NUMERIC '(' Prec ',' Scale ')' is an Exact decimal number with precision Prec and scale Scale. Prec must be between 1 and 18 (or 38 when HUGEINT is also supported). Scale must be between 0 and Prec.

- CHAR '(' length ')' is UTF-8 character string with length upperbound limit. CHAR or CHARACTER without the "(length)" specification is treated as CHAR(1).

"COPY INTO" can be used to import the file "earthquake.CSV" which has both the earthquake information into MonetDB.

```
COPY INTO  usgs FROM "C://earthquake.CSV"
      DELIMITERS ',','\n','"' NULL AS '';
```

In the code before $','$, $'\backslash n'$, $'"'$ are the functions to distinguish Comma, double quotes and skipping. "NULL AS" can be used to display the empty value. The execution time is 5273.97ms. There are 11237 tuples in this database.

5) Value query

By using the database with earthquake, searching all of the information with 'Source' in 'us' obtains 992 tupples in 22.344 ms.

```
select * from usgs where Source = "us";
```

2.2.4 Key-value Store database

As one of the type of NoSQL storage, the data of Key-value store databases [24] is organized, indexed and stored as key-value pairs. Key value storage is very suitable for the data which does not involve too many data relationships and effectively reduces the times for reading and writing from/into a disk. Hence, it has a better capacity of reading and writing performance than SQL database storage.

A key-value database[25] is a data storage paradigm designed for storing, retrieving, and managing associative arrays. Key value store is similar to this name, it contains a key and a series of values for this Key. The name of this key can be of any type. Because of this feature, the key-value store database

could quickly search the desired value. For example, Figure 2.14 presents an simple key-value database with the earthquake information. In this Figure, the values contain: 'time', 'latitude' and 'longitude'.

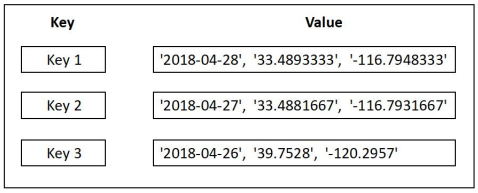


Figure 2.14: An example of key-value store database

Cassandra [25] [26] is not only a Key-Value database but also a distributed network service composed of numerous databases. For example, some reading or writing operation for Cassandra will be copied to a node in the network. Therefore, Cassandra has superior extensibility, as long as you add nodes to it.

Column concept

If there are three data are stored in a value pair, the storage address of each value must be recorded. Otherwise, the data could not be accessed. Therefore, Cassandra adds a name to each data, which is a way to find the data. Such a set of data containing name value pairs is called a row, and each set of name value pairs is called a column. It is the most basic data structure in Cassandra. It contains three data types, name and timestamp. Figure 2.15 describes an example of Column. The maximum storage of a Column name is 64KB. The value of the Column name cannot exceed 2GB. The name of the Column can be any data type. It is also possible to store the Column name as a data. The user can directly add data to Cassandra without being constrained by the data type. On the other hand, 'clock' in the figure records the time when the value was last modified.

Super Column concept

If the value in the Column is not a simple value and is split into multiple Columns, this large Column is called a Super Column. Both Column and Su-

Column		
key	value	clock: IClock

Figure 2.15: Model of column

perColumn are a combination of name and value. The biggest difference is that the Column's value is a "String", and the Super Column's value is a 'Map' consisting of multiple Columns. The SuperColumn itself does not contain a timestamp.

Column Family concept

The multiple column key-value pairs make up the Column family. The Column family is a container of columns. It is mainly used for logical segmentation and associates similar data. A Column family is similar to a table in a relational database. For example, Figure 2.16 presents a Column Family example with earthquake information in 'us'. Hence, the name of the Row key is 'us'. The Column key is the time that the earthquake happened. The Column value is the 'place' of earthquake.

us	2018-04-26	2018-04-27	2018-04-28	...
	9km NE of Aguanga, CA	22km ESE of Anza,CA	14km SSE of Mentone, Texas	
...				

Figure 2.16: Column Family of earthquake information

Keyspaces concept

Keyspaces is a container used to encapsulate Column families. At the same time, the keyspaces is Cassandra's data container. It can be understood as a table in a relational database. Therefore, Cassandra's space model can be divided into four or five dimensions.

The four dimension is:

The five dimension is:

```
set KeySpace1 . ColumnFamily1 [ 'rowkey1'] [ 'columnname 1'] = 'column value 1'

set KeySpace1 . ColumnFamily1 [ 'rowkey1'] [ 'columnname 1'] [ 'columkey 1']= 'column value 1'
```

Cluster concept

A cluster could contain multiple keyspaces. Therefore, it is the upper level of the key space.

Sorting rules

Cassandra sorts by column name and it supports the following data types: Characters, bytes, numbers and dates. Two data types in it are very important: bytes and characters. BytesType is the default sorting method for Cassandra that compares bytes directly without checking whether the content of the bytes conforms to a certain encoding. Its ranking rules are arranged from small to large. The second one is 'UTF8Type', mainly used for characters string. Its order is still ranked by the first character of the column name from small to large.

Distribution

Cassandra is a multi-node network service, and data is backed up at different nodes following the replication property. This parameter determines how newly-written parameters are copied and saved in each node. The replication property has a class and a replication factor which defines the kind of replication: SimplyStrategy (single data center), NetworkTopologyStrategy, etc, the amount of replication on offer. The durable write property governs the loss of data if a node suddenly stops working.

For example, the code below creates a Keyspace with the name "earthquake" and only one node to deal with the data.

```
create keyspace earthquake with replication =
       {'class':'SimpleStrategy', 'replication_factor':1};
```

Data design pattern

1) Row-Oriented

The data in Cassandra is stored in the database. Each row of data can have a different column structure. In a relational database, each row must have the same column. In Cassandra, it can use a unique ID to access the row. Therefore, Cassandra is an indexed, row-oriented storage.

2) Schema Free

Cassandra just defines a 'KeySpace' to load column families. This method is free to add data to the column family. Each column family is designed to be a set of data associations or data sorts. In this way, only the required data can be saved as necessary.

2.1.4.9 Sample

Cassandra uses the Cassandra Query language (CQL) [27] for data queries. Its full name is Cassandra Query Language. The CQL language is similar to SQL statements. It includes: modify, query, save, change the storage of data and so on. Each statement is terminated by a semicolon ';'. In general, there are three steps to creating a Cassandra database.

1) Creation Keyspaces

The rule [4] is: $CREATE\ KEYSPACE\ <identifier>\ WITH\ <properties>$;. The properties have two basic elements: Replication and Durable _ writes.

2) Creation Columnfamily

The rule for this approach is: $CREATE\ (TABLE)\ <tablename>\ '('<column-definition>','<column-definition>'\ ...)$;. In this code, defining the Primary key (Row key in Key-value store database) is very important. For example, the code given below describes the three values in Keyspace "earthquake". It defines the order in which the data is arranged.

```
CREATE TABLE earthquake.information
       (time  TIMESTAMP,
```

[4] https://www.datastax.com/wp-content/uploads/2013/03/cql_3_ref_card.pdf

```
        latitude REAL, longitude REAL, depth REAL,
        mag REAL, Source CHAR(2),
        PRIMARY KEY(time)
)
```

3) Insertion data

The rule for inserting data is: $INSERT\ INTO\ ' < tablename > '\ ' < columnname > '\ VALUES\ ('< values >')$; For example, an earthquake is inserted in the table using the following instruction:

```
INSERT INTO earthquake
      ( time, latitude, longitude, depth, mag, Source )
      VALUES
          ( 2018-05-14T11:48:15.020Z,
            -11.4242, 166.192, 71.4,
            4.6, us
          );
```

4) Value research

In this step, Cassandra has searched the earthquake information which has been happened in 'us'. By using the function 'Tracing on', it could also obtain the table for the execution time. There are 992 rows with 'us'. The execution time is 4.697ms.

```
Tracing on;
SELECT * FROM usgs WHERE Source="us"
ALLOW FILTERING;
```

2.2.5 Document database

MongoDB is a free and open-source cross-platform document-oriented database [8]. It can achieve high performance, high availability and also be easily extended. MongoDB does not need to define the table structure, however it could query data through complex query languages. The operational model for

MongoDB is mainly defined by two concepts: collection and document. For example, Figure 2.17 shows a database with one collection and two documents.

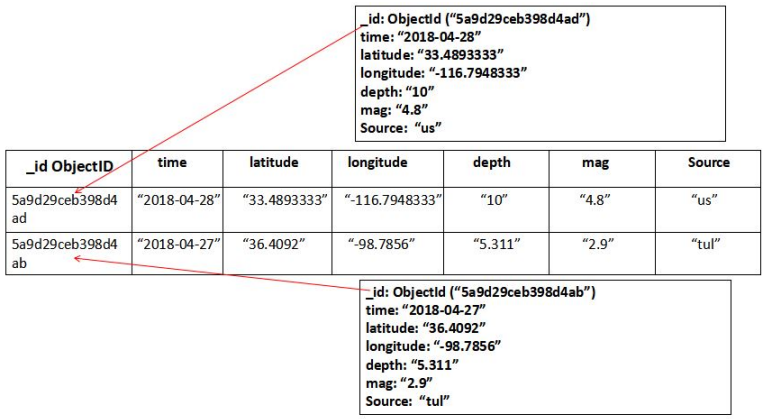


Figure 2.17: A simple database in MongoDB

Database concept

A single instance of MongoDB could hold lots of independent databases. Each independent database has its own collections.

Collection concept

A collection is a set of MongoDB documents. It is equivalent to the concept of a table in a relational database. A collection doesn't belong to a specific schema. Multiple documents within a collection could have many different fields and may be in the same field with different types. In general, documents in a collection have the same or related purpose. On the other hand, a Collection requires an id field which is considered as the primary key. Figure 2.18 presents the terminology of relational database and MongoDB.

There is a special collection: capped collections, which are a kind of limited buffer size very suitable for logging functions.

Relational Database	MongoDB
Database	Database
Table	Collection
Record	Document
Field	Field
Index	Index
JOIN table	Embedded documentation
Primary key	Primary key (Default key id is offered by MongoDB)

Figure 2.18: Terminology of relational database and MongoDB

```
db.createCollection("log", {capped:true,size:100000})
```

Primary key

MongoDB uses a ”collection” which is similar to a table and ”documents” which are similar to rows, to store the data and schema information. All documents in a MongoDB collection have a PRIMARY KEY [9] called $\_id$. This field is automatically assigned to a document upon insertion, so it rarely needs to be provided.

For example, the addition of a user into the ‘user’ collection from the MongoDB database can be obtained with the following commands.

```
db.earthquake.insert({
    "time":"2018-04-28",
    "latitude": 33.4893333,
    "longitude": 166.192,
    "depth": 71.4,
    "mag": 4.6, "Source": "us"}
);
```

Index

Index can usually improve the efficiency of the query. If there is no index, MongoDB must search each document in the collection and select the records that meet the query conditions. Index are special data structures. They are stored in a set of data that is easily traversed and read. An index is a structure

that stores the values of one or more columns in a database.

Document concept

A Document is a set of key-value pairs. Documents have a dynamic pattern which means that documents in the same collection do not need to have the same fields or structures. An example of inserting a document is given as follows :

```
db.earthquake.insert(
{ "time": "2018-04-28",
  "latitude" : 33.4893333,
  "longitude" : -116.7948333,
  "depth": 71.4,
  "mag": 4.6, "Source": "us"
})
```

Embedded Document

An embedded document is when a document is embedded within another up to a limit of 16MB. A document, including all its embedded documents, cannot exceed 16MB. The code below is an example of an embedded document.

```
{ type: 'earthquake',
informations: [
               {"Source": "us",
                "time":"2018-04-28",
                "latitude": "33.4893333",
                "longitude": "-116.7948333",
                "depth":"71.4",
                "mag":"4.6",
               },
               {"Source": "tul",
                "time":"2018-04-28",
                "latitude": "-6.1579",
                "longitude": "142.4845",
                "depth":"5.311",
                "mag":"2.9",
               }
```

```
                    ]
    };
```

Join operation

MongoDB database can deal with unstructured data as illustrated in Figure 2.19.


```
{
        "_id" : ObjectId("5afeed6379feb04ec5ca68e4"),
        "time" : "2018-05-14T14:09:02.150Z",
        "latitude" : 19.3348331,
        "longitude" : -154.9918365,
        "depth" : -0.68,
        "mag" : 1.91,
        "magType" : "md",
        "id" : "hv70155912",
        "type" : "earthquake",
        "Source" : "hv"
}
```


Figure 2.19: Examples of document

The key can be used to identify a particular data item and the value could be words, numbers or semantics structures. Therefore, to query the collection of objects in MongoDB is based on its PRIMARY KEY which makes MongoDB similar in its behavior and purpose to a SQL JOIN.

However, MongoDB stores denormalized data and there is no relationship between collections: if there are the same data required in two or more documents, it must be repeated. MongoDB introduces an operator $lookup$ which can perform a LEFT-OUTER EQUI-JOIN to get all the documents from the left collection together with data from the right collection where there is a match with the left collection. Nevertheless, $lookup$ is only permitted in aggregation operations as illustrated below:

```
db.leftCollection.aggregate
[{($lookup:
{from: "rightCollection",
 localField:"leftVal",
 foreignField:"rightVal",
 as: "embeddedData"}
}])
```

The aggregation pipeline is a framework based on the concept of data processing pipelines. By using a multi-stage pipeline, a set of documents is converted to the final aggregated result. There is an example below to choose the earthquake that happened in ”us” and which was written in the domain of ”Earthquake”. Figure 2.20 presents this example.

```
db.orders.aggregate ([
                        { $match:
                          { Domain: "Earthquake"}
                        },
                        { $group:
                        { Source, total:
                        { $sum: "$Earthquake"}
                        }
                        }
                      ])
```

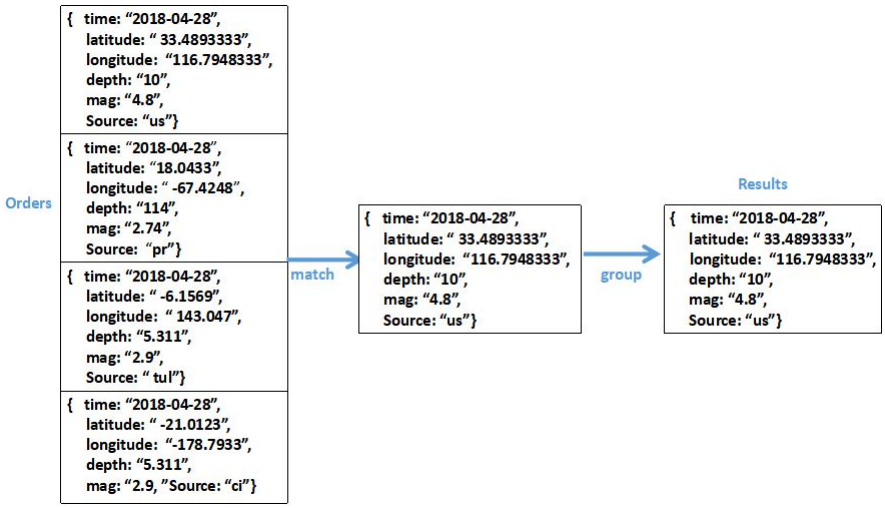


Figure 2.20: Aggregation operation in MongoDB

BSON

MongoDB uses BSON documents to store records. BSON is based on the Javascript object Notation [23] (JSON) which is similar to XML but smaller,

faster, and easier to parse. The JSON text format is syntactically the same as the code that creates a JavaScript object. Because of this similarity, no parser is required, and JavaScript programs can use the built-in ”eval” function to use JSON data generates native JavaScript objects. There are two types of structures:

Object: An object contains a comma separated collection of non-sorted key/value pairs. The symbol of an object is { }

Array: This is a collection of values. The symbol of an array is [ ].

The specific format is as follows:

Name/value (pair): The name and value are separated by a colon. The general form is:$\{name : value\}$

BSON stores documents like JSON but in a binary and compact way. At the head of a document, it adds the size of the document, which allows a faster traversal speed.

BSON will mainly achieve the following three goals:

1) Faster traversal speed: Too large JSON structure causes very slow data traversal. In JSON, to skip a document for data reading, the document needs still to be scanned. BSON’s improvement over JSON is that it will store the length of each element of the JSON at the head of the element. In this way, it only needs to read the length of the element to directly find the specified point for reading.

2) Easy to operate: Data storage is type aware. For example, if there is a database which wants to modify a basic value from 9 to 10. The size of the field has changed from one character to two, and all the contents behind need to be one shifted. With BSON, it could specify this column as a numeric column. Therefore, whether the number is from 9 to 10 or 100, it will only make changes in the one where the numbers are stored, and will not cause the total length of the data to become larger.

3) Added extra data types: JSON is a very convenient data exchange format, but its type is limited. BSON adds the ”byte array” data type. This means that binary storage no longer needs to be Base64 [30] converted and

stored as JSON. This significantly reduces computational overhead and data size. BSON has no spatial advantage over JSON. For example,$\{"field" : 7\}$, which has only one byte is used in the storage of JSON. By using BSON, it has at least 4 bytes (32 bits). BSON provides some additional data types: Binary data, Floating point, Date and Timestamp. MongoDB has 18 different data types presented in Figure 2.21.

String	Null
Integer	Symbol
Boolean	Date
Double	Object ID
Min/Max keys	Binary Data
Array	Code
Timestamp	Regular expression
Object	

Figure 2.21: Different data types in MongoDB

Example

1) Insert document

Importing with CSV file with earthquake information recorded by the United States Geological Survey. It has imported this file into MongoDB. Then, it could see the size of this database and one item of the information in this database. The size of this database is 0.001GB. An example is:

```
mongoimport
            --db earthquakes
            --collection earthquake
            --type csv
            --file earthquake.csv
            --headerline
```

Figure 2.22 presents one of the result of earthquake information and the size of database ”usgs”.

2) Query

An example to find all of the information of earthquake in ”us” and the execution time. And obtain the execution at the same time. The execution time

is 0.01s. Figure 2.23 describes the information obtained and the execution time to realize this work by using find( ) to search the information directly. ”executionTimeMillesEstimate” is the result of execution time by using the unit ”ms”.

```
db.earthquake.find({"Source" : "us"}).
  explain("executionStats")
```


```
{
        "_id" : ObjectId("5afeed6379feb04ec5ca68e4"),
        "time" : "2018-05-14T14:09:02.150Z",
        "latitude" : 19.3348331,
        "longitude" : -154.9918365,
        "depth" : -0.68,
        "mag" : 1.91,
        "magType" : "md",
        "id" : "hv70155912",
        "type" : "earthquake",
        "Source" : "hv"
}
```

```
admin   0.000GB
local   0.000GB
test    0.003GB
usgs    0.009GB
```


Figure 2.22: One of the earthquake information and size of database


```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 992,
        "executionTimeMillis" : 19,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 11236,
        "executionStages" : {
                "stage" : "COLLSCAN",
                "filter" : {
                        "Source" : {
                                "$eq" : "us"
                        }
                },
                "nReturned" : 992,
                "executionTimeMillisEstimate" : 10,
                "works" : 11238,
                "advanced" : 992,
                "needTime" : 10245,
```


Figure 2.23: Execution time and all of the results with ”us”

On the other hand, Index could be used to optimise the execution time. The code below is to change the order of earthquake information in ascending order. Then, the execution time should be smaller than before.

```
db.earthquake.createIndex({"Source":1})
```

2.2.6 Graph database

A graph database is another kind of NOSQL database and as the name implies it uses graphs to store data or it stores data using triples organized in a graph way. There are two basic elements in a graph: Nodes and Edges between nodes. Both of them contain attributes in the form of key/values. A graph can also be expressed as a parametrized data type : $G(N, R)$, G is a graph, N is the type of nodes and R is the type of Relations. Figure 2.24 describes an example of relations in a graph. In this Figure, there are some labels between each node. This is a graph labeling. It is the assignment of labels, traditionally represented by integers, to the nodes or relations, or both of a graph. The node could also occupy some properties.

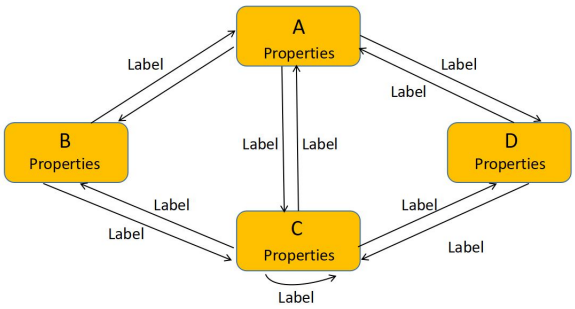


Figure 2.24: The basic attribute of a graph

Node and Edge concept

If there is a direction from the relation of Node1 to Node2, the Edge is said to be directional.

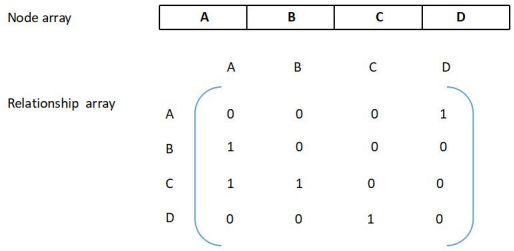
By using the Edges, NEO4J could obtain lots of associated data, such as the collection of nodes, collections of relations and their collection of attributes. On the other hand, a node can have an Edge that points to itself. And there are also many Edges between the same nodes.

InDegree and OutDegree

In a directed graph $G = (N, \{R\})$[29], if there is an Edge (N, N'), N and N' are Adjacent. The node (N, N') is attached to nodes N and N',or (N, N') is associated with N and N'. The degree of Node is the number of edges associated with Node. The number of arcs headed by node N is called the InDegree of N. The number of arcs headed by node N is called the InDegree of N (ID (N)).The number of arcs tail by node N is called the OutDegree of N (OD (N)). The degree for this Node is TD(N) = ID (N) + OD (N)

Adjacency Matrix

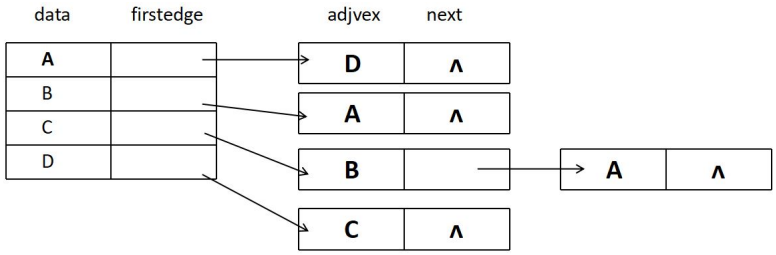
In general, an adjacency matrix is represented by two arrays. a one-dimensional array stores "Node" information in the graph and a two-dimensional array (Adjacency Matrix itself) stores information about nodes or edges in a graph. There is an example below to explain the storage by using the graph.



The node array is node[4]= $\{A, B, C, D\}$. The Relationship array is "Relation"[4]. The degree for Node C is 2. The degree for the other nodes is 1. For example, if there is an arc from B to A then $relation[1][0] = 1$. However, there is not a relation from A to B, the result is changed to: $relation[0][1] = 0$. Hence, the discrimination approach to search the relationship between two nodes is $relation[i][j] = 1$. NEO4J asked all of N's adjacencies to scan the elements of the i-th row of the matrix and find the nodes whose $relation[i][j]$ is 1.

Adjacency List

Adjacency List is another approach to store graph. There is an example below to explain the storage by also using the graph. In this example, the Node table contains two fields, data and ”first edge”, ”Data” is the data field and ”first-edge” is the pointer field. ”Firstedge” points to the first node of the side table. ”Adjvex” is the adjoining field. ”Next” points to the next node in the side table. Therefore, it can also calculate the degree and search the relationship between different nodes. In NEO4J, the query will be realized by searching the Adjacent node. The query of NEO4J is based on the node. It has searched all of the other nodes which have a direct relation between this node.



Join operation

Join operation for graph database is a selection over a Cartesian product. A Cartesian Product is a collection from multiple subgraph. The data model of Cartesian Product is : $A \times B = \{(a, b) \mid a \in A \ and \ b \in B\}$. A and B are the nodes. ”a” and ”b” are properties.

Create Nodes

NEO4J [28] is a kind of graph database mainly used in ontology representation so it respects a set of standards such as the Resource Document Framework (RDF) [15] itself based on XML format. NEO4J defines the Cypher language

to query the database. In NEO4J graph database, it has used Cypher language[5] to query the database.

```
CREATE (a:Earthquake{"time":"2018-04-28",
                     "latitude": "33.4893333",
                         "longitude": "-116.7948333",
                     "depth":"71.4",
                     "mag":"4.6",
                     "Source": "us"}
           )
CREATE (b:Earthquake{"time":"2018-04-27",
                     "latitude": "18.0433",
                     "longitude": "-67.4248",
                     "depth":"114",
                     "mag":"2.74",
                     "Source":'pr'}

       )
```

In the example above, the node "earthquake" is enclosed in round parenthesis and defined by several properties, comma separated, represented by edges to value all enclosed in curly braces.

```
(a:Earthquake{"time":"2018-04-28",
              "latitude": "33.4893333",
              "longitude": "-116.7948333",
              "depth":"71.4",
                  "mag":"4.6",
              "Source": "us"})
(b:Earthquake{"time":"2018-04-27",
              "latitude": "18.0433",
              "longitude": "-67.4248",
              "depth":"114",
              "mag":"2.74",
              "Source":'pr'})
```

All of the codes before are "adding" Nodes. There are two different nodes here. In the code before, the label is "Earthquake".

[5] http://people.inf.elte.hu/kiss/13kor/Neo4jCheatSheetv3.pdf

```
a:Earthquake{"time":"2018-04-28",
             "latitude": "33.4893333",
             "longitude": "-116.7948333",
             "depth":"71.4",
             "mag":"4.6",
             "Source": "us"}
b:Earthquake{"time":"2018-04-27",
             "latitude": "18.0433",
             "longitude": "-67.4248",
             "depth":"114",
             "mag":"2.74",
             "Source":'pr'}
```

All of the codes before are the properties of each node. It has used " " square brackets to define the properties of a node.

Create a Relation

```
MATCH (a:Earthquake),(b:Earthquake)
WHERE a.Source = "us" AND b.Source = "pr"
CREATE (b) - [:R] -> (a)
RETURN type(r)
```

- "MATCH" is the way to get data from the graph. This is the graph pattern to match. - "CREATE" creates nodes and relationships. This part describes the realtion between a ("us") and b ("pr") by ":R".

Querying style

```
MATCH (x) - [:R] -> (y)
WHERE y.Source = "us"
RETURN x.Source
```

The query ask to search the other earthquake's source which has a relation ":R" with "us". Figure 2.25 describes the construction of this sample of graph DB.

```
CREATE (a:Earthquake{"time":"2018-04-28",
```

```
                          "latitude": "33.4893333",
                        "longitude": "-116.7948333",
                        "depth":"71.4",
                        "mag":"4.6",
                        "Source": "us"}),
CREATE (b:Earthquake{"time":"2018-04-27",
                       "latitude": "18.0433",
                       "longitude": "-67.4248",
                       "depth":"114",
                       "mag":"2.74",
                       "Source":'pr'}),

((b) - [:Source] -> (a));
MATCH ((x) - [:Source] -> (y))
WHERE y.Source = "us"
RETURN x.Source;
```

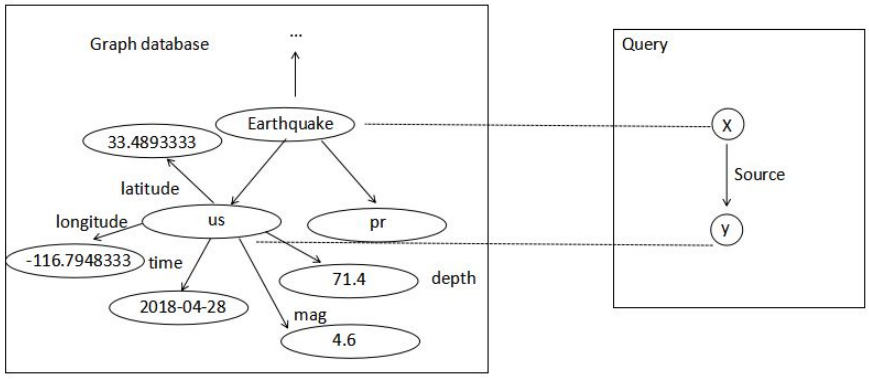


Figure 2.25: Sample of graph DB

Example

1) Nodes creation.

```
CREATE (p:test{
        time:line.time,
        latitude:line.latitude, longitude:line.longitude,
        depth:line.depth, mag:line.mag, Source:line.Source
```

```
        }
    )
```

2) Values query

- NEO4J could search all of the information by using the Source name. In this database, it's name is 'Source'. There are 15 countries and 11252 earthquakes in the world. It has used 3491ms to search all of these information. For each Source there are the relations to connect with any earthquake information.

```
MERGE (c:Source {code:coalesce(line.Source, 'NA')})
CREATE (t:Test{
               time:line.time,
               latitude:line.latitude,
               longitude:line.longitude,
               depth:line.depth,
               mag:line.mag,
               magType:line.magType,
               id:line.id,
               type:line.type,
               Source:line.Source
               }
      )
MERGE (t)-[:LOCATED_AT]->(c)
```

- NEO4J could query the same question: how many times did it should use to search the earthquake information in 'us'? It has used 307ms. There are 992 nodes. It means that there are 992 earthquakes in US.

```
WHERE line.Source='us' WITH line
MERGE (c:Source {code:'us'})
CREATE (t:Test{
                    time:line.time,
                    latitude:line.latitude,
                    longitude:line.longitude,
                    depth:line.depth,
                    mag:line.mag,
                    magType:line.magType,
```

```
                    id:line.id,
                    type:line.type,
                    Source:line.Source
                }
        )
MERGE (t)-[:LOCATED_AT]->(c)
MATCH (c) RETURN c
```

2.2.7 Summary

Relational databases and NoSQL databases are not opposed to each other, but are complementary.

Data models

Each database or data service network has its own fixed data type. The following table compares the data types of the five databases. Figure 2.26 describes the database information.

Type	RDBMS	Column	Key value	Document	Graph
Name	Sqlite	MonetDB	Cassandra	MongoDB	NEO4J
Database structure	Table	Table	Cluster	Collection	Nodes
	Record	Row	Key space	Document	
	Field	Column	Column Family	Field	
	Index	Index	Super Column/Column	Index	Relations
JOIN Approach	JOIN table	JOIN table	JOIN column	Embedded documentation	JOIN node and Relation
	Primary key	Ids	Column key	Primary key	Adjacency node
Schema	Schema	Schema	Schema free	Schema free	Schema free

Figure 2.26: Databases information

Relational databases

In relational database, the index which is unique and if the request does not relate to this index (which is often the case) the search will be slow. Making an optimal database in space supposes many tables and unfortunately for the processing time it will be necessary to make a join which can be slow if one

does not operate on primary keys and which can become catastrophic if one distributes on several servers.

Column-oriented databases

Due to the explosive growth of data volume in recent years, this type of NoSQL database has attracted special attention. Normal relational databases store data by units of row, and are good at reading by row, such as the acquisition of specific conditional data. Therefore, relational databases have been named as ” row-oriented databases”. In contrast, a column-oriented database stores data in units of columns and is good at reading data in columns.

Column-oriented databases have stability: if the data is increased, it will not reduce the corresponding processing speed (especially the writing speed). So they are mainly applied where a large amount of data needs to be processed. In addition, they are also very useful to update large amounts of data for a batch program. The indexes on the data are cascaded so that an optimal construction of the base could give very high performance if the queries are known during the construction. However, due to the fact that column-oriented databases are very different from current database storage, they are very difficult to apply. Figure 2.27 presents the difference between RDBMS and Column store database.

Type	Storage	Advantage
RDBMS	Rows as unit	Read and update a small number of rows
Column store	Columns as unit	Reading a large number of rows for a few columns, updating all the rows in a specific column at the same time

Figure 2.27: Comparison between RDBMS and Column store database

Key-value store and Document databases

Key-value store databases are suitable for applications that operate frequently and have a simple data model. The values stored in the key-value database can be simple scalar values, such as integers, booleans, or they can be structured

data types, such as lists and JSON structures. A key-value database usually has a simple query function that allows us to find a value by key. General key database support search functions, provides greater flexibility. Developers can choose to use techniques such as enumerated keys to implement range queries, but these databases often lack the ability to query documents, column families, and graph databases.

Document databases provides embedded document, which is useful for denormalization. The document databases stores frequently queried data in the same document, not in different tables. The document database is designed according to the standard of flexibility. If an application needs to store different attributes and a large amount of data, the document database will be a good choice.

Graph databases

1) The traversal of a graph is a unique algorithm of the graph data structure, that is, starting from a node. It can quickly and easily find its neighboring nodes according to its connection relationship. Therefore, Neo4J has very efficient query performance, which can increase the query speed by several times or even dozens of times compared to RDBMS. And the query speed will not drop due to the increase of data volume, that is, the database can be durable and always maintain its original vitality. Unlike RDBMS, some of the paradigm designs are inevitably used, if the developer needs to represent complex relationships during a query, a lot of connections will be inevitably constructed.

2) The characteristics of the graph data structure and its unstructured data format make the Neo4j database design highly scalable and flexible. Because the nodes, relationships, and attributes that increase as the demand changes do not affect the normal use of the original data. In fact, the properties of Neo4j nodes are some Key-Value data collections. Neo4j can display richer contents through the attributes of nodes and relationships, which is incomparable to other Key-Value databases.

Schema and Schema free

Some database are schema based (Sqlite, MonetDB) and others are schemaless (Cassandra, MongoDB, NEO4J). If the user should understand and define the structure of a database then it is a schema-based database. The relational schema defines what columns appear in the table, their names, and their data types. However, it is an error to insert data that doesn't fit the schema. Therefore, the scalability of the database is not high.

In contrary, a schemaless database allows any data, structured with individual fields and structures, to be stored in the database. One of the most commonly used guidelines here is that schemaless can benefit the user with this flexibility. For example, compared to schema database, schemaless database could bring more development flexibility to software developers, but their maintainability and rigor are often not as strong as typed languages. Similarly, flexibility and maintainability should also be taken into account when using a schemaless database. In this way, schemaless database could add a variety of relationships to the nodes, instead of worrying about whether to record some foreign keys by changing the schema of the database, as in a relational database. This in turn allows software developers to add a variety of relationships between nodes.

Execution time with different database

To give a first idea on performance, we load each database with earthquake information provided by the United States Geological Survey for one month. Figure 2.28 shows the different execution time by searching same query "The earthquake information located in "us". All the execution times are obtained by a query not using an index.

Database	Sqlite	MonetDB	Cassandra	MongoDB	NEO4J
Execution time(ms)	4	22.344	4.6997	10	307

Figure 2.28: Sample information

In Figure 2.28, the execution time is good for three databases. ”Sqlite”, ”MonetDB” and ”MongoDB”. The reasons why it causes this result are:

1) Cassandra and MongoDB are schemaless databases and schemaless networks. There are also Column key and Primary key in these two databases. The time to search the relevant values in a database is fast. However, Cassandra is three times much more faster than MongoDB, because of the data model for Cassandra is based on a collection of Column family. For some Column family has the same row key. For example, Cassandra could use ”us” for the Row key.

2) This earthquake database is a file with lots of data types. Structurally it is near to a relational database and as a relational database, Sqlite could obtain the result a little faster than Cassandra and 3 times faster than MongoDB.

Therefore, based on the data in the Figure 2.28, there are some following conclusions: Whether it is a schema database or a schemaless database. NoSQL database or SQL database. They all need to analyze and select the appropriate database according to the specific situation.

2.3 Category theory

Category theory [35] [36] uses functions to express the relationship between categories. With the development of category theory, a set of calculation methods for functions has been developed. This theory was originally used only for mathematical operations, and later it was implemented on functional programming. Haskell is a functional programming so it is based on category theory.

Category was introduced by an American mathematician Mr. Mac Lane [40]. It is the ultimate abstraction of most mathematical objects. Abstraction refers to the process of describing different concepts in the same way. For example, all sets, linear spaces, groups, and graphs form a category. Category Theory includes three basic elements: Category, Functor and Natural Transformation. In the other hand, there is also the Monad endofunctor.. Category theory generalizes many fields of computer science such as Algebraic specifi-

cation[41], program calculation [42] and Model Driven Engineering [43].

Definition

In life, people often say: "They are not talking about something in a category!" The implication is that the things that two people talk about are not related and have no relevance. In fact, Category refers to a group of things and all the relationships between these things. In other words, these things and these associations together constitute a category. In general, Category theory is a mathematical theory that abstractly deals with the relationship between mathematical structure. In order to define this work, it has three basic elements. The objects with some functions, functors which describe how to switch objects between two categories and natural transformation to describe ways to apply functors.

Benefits and Uses

The usefulness of Category theory is to provide a more unified and a more general representation of mathematical structures. The French mathematician group: "Nicolas Bourbaki" summarized three basic mathematical structures: algebra, topology and order into one Category theory. Any currently mathematical structure can be expressed in the language of category theory. At the same time, the graph of categories is also very intuitive and easy to understand.

2.3.1 Category

Definition

A category is a mathematical structure [33][34] $C = (O, A, 1, o)$ defined by: a set of Objects, e.g $A, B, C, ...$, a set of Arrows (morphisms) "A" between theses objects. An application that associates a source and target object to each arrow $f : A \to O \times O$. For this relation, an arrow "a" such as $f(a) = (O_1, O_2)$ is simply written $a : O_1 \to O_2$. It is pronounced by a is the type of O_2 from O_1. Figure 2.29 describes the example with the composition and identity relation.

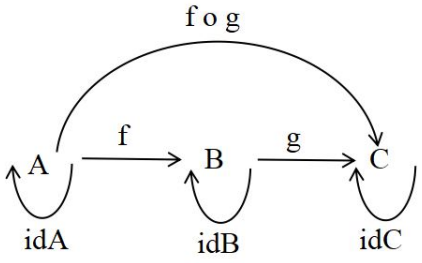


Figure 2.29: An example with the composition and identity relation

Figure 2.29 describes an example based on three objects (A, B, C) with two morphisms $f : A \to B$ and $g : B \to C$, identity morphism $ido : O \to O$ and a composition operator $f \circ g$.

Properties

A category is generally represented by a directed graph such as the one of the Figure 2.30.

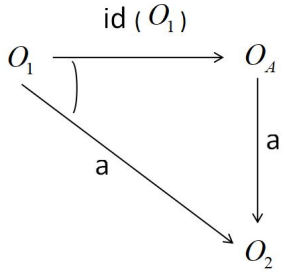


Figure 2.30: A directed graph

A diagram is commutative if two paths from a common object lead to the same element. In particular, Figure 2.31 shows a commutative diagram in the sense $a \circ id(O_1) = a$. Therefore, this kind of diagram corresponds to an equation (and also corresponds to a program in a functional programming language).

Categories are associated with concepts such as

- Duality and opposite category (obtained by reversing the arrows)
- Initial object (or by the way of duality, final object), limits/colimits, pushout/-

pushback. In particular, a pushout for a diagram ($X \overset{g}{\leftarrow} Z \overset{g}{\rightarrow} Y$) is an object P associated with two morphisms ($X \overset{i_1}{\leftarrow} P \overset{i_2}{\rightarrow} Y$) such as the following diagram commutes:

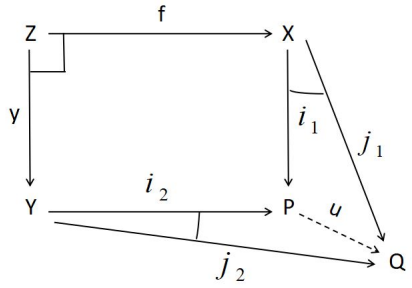


Figure 2.31: Commutative diagram

The pushout is associated with the universal concept which means that for the other pushout (Q, j_1, j_2) there exists an unique morphisms $u : P \rightarrow Q$. In other words, the pushout is unique up to the iso-morphism.

Examples

1. Set

A simple Category is Set whose objects are all sets and arrows are partial functions. Pullbacks correspond here to disjoints union, and pullbacks to cartesian products. Products are written $X \times Y$ and are associated with projections functions $\pi_1 : X \times Y \rightarrow Y$ and $\pi_2 : X \times Y \rightarrow Y$. Figure 2.32 presents a diagram with Products and Pullbacks.

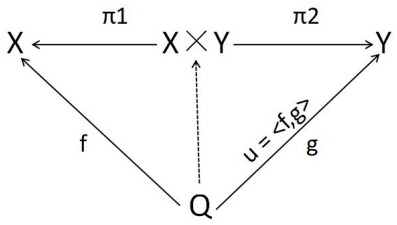


Figure 2.32: Products and Pullbacks

2. Graph

The objects of Graph are graphs and morphisms are a graph morphism. An illustration of a graph morphism is given in Figure 2.33.

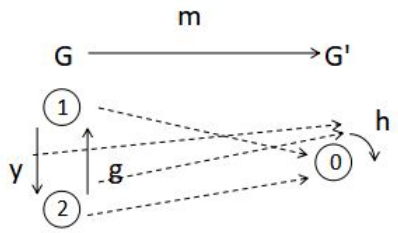


Figure 2.33: Example of graph morphism

It reminds us that a graph morphism is given by a pair of applications (one to transform the node and the other to transform edges) preserving the structure of the graph. As a remark, the graph G' in Figure 2.33 is an example of a terminal object.

Sample

A graph can be used to represent a relation and $(x_1, x_2) \in G$ can be interpreted as "x_1 is related to x_2". The logical expression: $\exists X, (X, 1) \in G \wedge (X, 4) \in G$ expresses the search of the common friends between 1 and 4 that is graphically represented by the graph G. Figure 2.34 describes an example of graph.

2.3.2 Functor

Defintion

A functor F is a structure preserving map between two categories $F : \mathscr{C} \to \mathscr{D}$. These means that it transforms objects and arrows. If $a : O_1 \to O_2$ is an arrow from $\mathscr{C}$ then $F(a) : F(O_1) \to F(O_2)$ is an arrow from $\mathscr{D}$. The functor also presents identity and composition. For example, $F(a_1 \circ a_2) = F(a_1 \circ F(a_2)$ and $F(id(a_1)) = id(F(a_1))$

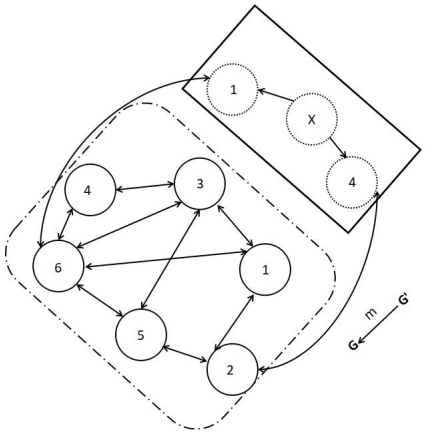


Figure 2.34: An example of graph

Endofunctor

In the category Set, a well-known (endo)functor is the Powerset such as $\mathscr{P}(0)$ correspond to the set of all the subset of "0" and $\mathscr{P}(f)\{x_1...x_n\}$
$= \{f(x_1)...f(x_n)\}$.

Example

Category is the product (bi)functor such as $O_1 \times O_2$ represents the set of pairs (x, y) and $(f\ x\ y)(x, y) = (f(x), f(y))$. The preceding functors could build new ones and be used to abstract data structures found in computer science. For example, $\mathscr{P}(x \times y)$, $\mathscr{P}(x \times \mathscr{P}(y))$. More generally, (abstract) datatypes such as List or Tree, could be defined indirectly by a set of morphisms (operaters) and as the fixpoint of a functor.

Sample

As an illustration in Figure 2.35,List L(x) could be defined by a function "e" which returns the empty list and a function "a" to add an element of type x. The "1" represents the initial object from the category and corresponds both to any singleton set or the "void" datatype in a programming language.

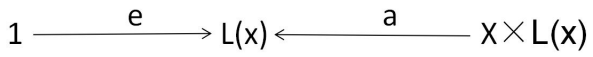
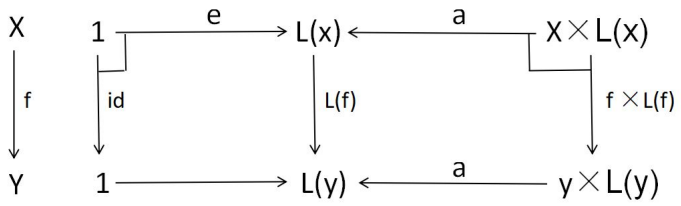


Figure 2.35: Example of list

Then, L(f) is defined by the commutative diagram:

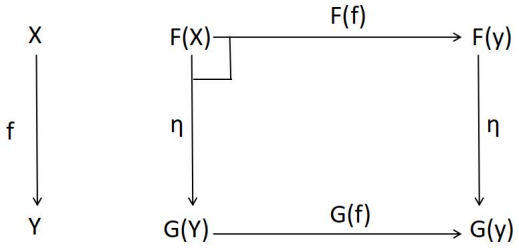


Thus, a sample list [1,2,3] will be represented by $a(1, a(2, a(3, ls))$ and one can check that, with the preceding diagram or equation, $L(f)(a(1, a(2, a(3, ls))))$ $= a(f(1), a(f(2), a(f(3), ls) = [f(1), f(2), f(3)]$. As a remark, L(f) is also called "map(f)" and is also the most well known of the functional programming users.

2.3.3 Natural transformations

Definition

In Category theory, Natural Transformation $\eta : F \to G$ is an important concept in dealing with the relationship between two functors F and G such as for any object x and y the following diagram commutes :



Example

Natural transformations are interesting to relate or transform data structures, and a concrete example is $\eta : \mathscr{P}(x \times y) \rightarrow \mathscr{P}(x \times \mathscr{P}(y))$ that returns the adjacents elements. For example, $\eta\{(x_1, y_1), (x_1, y_2)...(x_n, y_n)\}$ $= \{(x, \{(x, \{y_1, y_2...\}), ...(x_n, \{...\})\}$. The fact that there exists an inverse transformation $\eta' : G \rightarrow F$ such as $\eta' \circ \eta = G$ and $\eta \circ \eta' = F$ leads to the concept of "isomorphism" or equivalence as illustrated in the following Figure 2.36.

Intuitively, by considering the example above in Set, the two data structures $(\mathscr{P}(x \times y)$ and $\mathscr{P}(x \times \mathscr{P}(y))$ represent a "same" information and are naturally isomorphics. As a remark, in the context of computer program, the algorithms (eg. get the adjacents elements) mostly depends on the data structure considered and then natural transformation could be considered to "shift" program to another data structure and get better performance (eg. algorithm optimization).

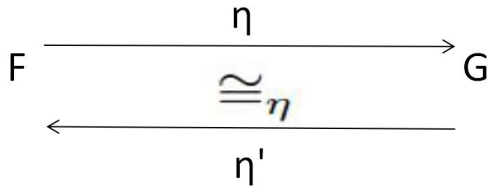


Figure 2.36: Example of natural transformation

2.3.4 Monad

Definition

In functional programming languages, Monad [44] is a very important concept. It originated from the Category theory in mathematics. It is a common property of a type of type. We can abstract some of the data-specific continuous computational behaviors and automate them. That is to say, the programmer does not have to care about what happens inside Monad when writing the program, but just use some basic operations of Monad, such as return and $\gg=$. Therefore,

Monad can be understood to some extent as the integration and consolidation of code. It can reduce the programming of duplicate and redundant codes.

Monads introduction

In computer science, the side effect of a function occurs when calling a function. In addition to returning function values, it also has an additional effect on the called function. For example, modify global variables (variables outside the function) or modify parameters. The ”side effects” of the function can cause unnecessary trouble to the program design, bring very difficult errors to the program, and reduce the readability of the program. However, Monad provides a good mechanism for handling ”side effects”.

Operating mode

In general, there are two basic elements in Monad: Value and Type.

1. Value

1) Value is the most basic data in the software A series of operations that handle Value can be encapsulated into functions. Entering a value will result in another value. For example $(+1)2 = 3$

2) Functions can also be used in conjunction, one function followed by another. At the same time, it can also process each member of the data collection in turn.

2. Type

A data type is a package of values that includes not only the value itself, but also related properties and methods. For example, After 2 becomes the data type, the original function cannot be used. Because the ”(+1)” function handles values (referred to as ”value functions”) rather than dealing with data types.Therefore, it needs to redefine an operation. It accepts a ”value function” and an instance of the data type as input parameters, and then uses the ”value function” to process another instance of the data type. Therefore, in fact, the

data type of the package is opened, the value is fetched, processed by the value function, and then the data type is encapsulated. For example, there are two types to calculate. Firstly, taking out the respective values, one is a function and the other is a Value. Secondly, using the function to process the value. Finally, the result of the function is then encapsulated into the data type.

Therefore, Monad is using this design pattern to split an operation process into multiple connected steps through a function. As long as the user provides the functions needed for the next operation, the entire operation can be automatically carried out.

Advantages

Monads [45] are interesting in a functional programming context to model "side effects" (eg. IO, Concerning, Non Determinist). As a simple example, the State monad that models the concept of "memory" found in imperative programming can be represented by $F(x) = S' \rightarrow X \times S'$. In particular, saving a value into the "memory" will correspond to $save(v) = ((\ ), v)$ and loading the in memory value to $load'(s) = (s, s)$. The return function then simply corresponds to $return(x) = \lambda s.(x, s)$ where the notation $\lambda x.y$ denotes an anonymous function $f(x) = y$. The element μ is replaced by an equivalent form called bind: $F(X) \rightarrow (X \rightarrow F(Y)) \rightarrow F(Y)$. This one simply corresponds to a "composition" of effects. Its definition is bind $m\ f\ = \lambda s$. The operator is associated with a specific notation in functional programming and bind m f is written:

```
do x -> m
return (f(x))
```

Thus, with all these elements, it is possible to express kinds of imperative programs such as:

```
do save (1)
v <- load
save (v+1)
```

2.4 Data and Querying Modeling

Big Data is described by five basic elements [46][53]: Volume, Velocity, Variety, Veracity and Value. However, only the two first elements are relevant for this research. Volume is about the amount of data. Velocity proposes a highly growth rate of data in Big Data. The research is concentrated on how to reduce the querying time by using a better storage structure. In our example, we query the earthquake database to find all records with a source from the united stats (US). Modifying the data model is an effective way to abstract the data values. As described before, Category Theory could be used to define and modify the data model to optimize the efficiency without influencing the working quality. For example, Functor defines the storage structure of data, Natural transformations could safely change Data modeling approach. Hence, the time processing could get a certain degree of improvement.

2.4.1 Time complexity

In computer science, an algorithm [47] is a description of the solution to a particular problem. It appears as a finite sequence of instructions in a computer, and each instruction represents one or more operations. The algorithm is not unique, that is, the same problem can have multiple algorithms that solve the problem. However, how can we improve execution time of the algorithm?

When performing algorithm analysis, the total number of executions of the statement T(n) is a function of the size of the problem n. Then, analyzing the change of T(n) with n and determine the order of T(n). The time complexity of the algorithm, which is the time metric of the algorithm, is denoted as: T(n) = O(f(n)). In our research, the time used to analyze the data relies on the number of values in a database. There is an example in Figure 2.37. The time to realize the operation with "$get(\ )$" and "$has(\ )$" is O(n). Because of the number of values in the list is "n". In this figure, "add x" is the function to add the element in a list. We set the time complexity of adding an x is O(1), the time complexity of our work will be O(n) by inserting n times of x. In

Figure 2.38, there are two equations. They have been used to verify the value (”time” = = y) in this database. If there is an element y in list L(x), it will be directly obtained the result of ”Bool”. On the contrary, DB contains a large amount of data information, the computer needs to spend more time (Compare data information for each item in the database) to complete the work when performing the same data retrieval task.

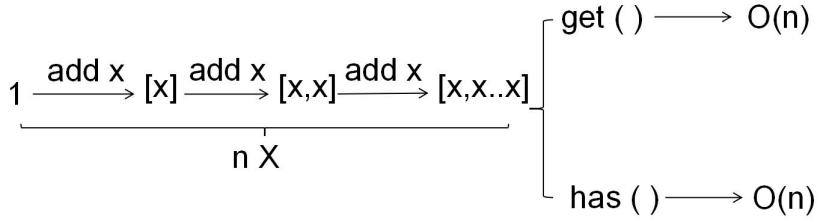


Figure 2.37: Schema for time complexity

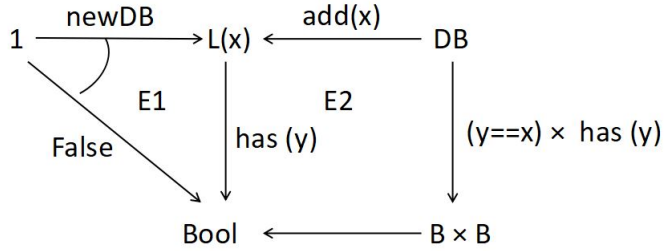


Figure 2.38: Example with two equations for time complexity

For example, the two equations in the figure before are:

```
E1 is  False = newDB o has (y)

E2 is  has(y) $\circ$ add(x) = (x==y) v o has (y)
```

Illustration, there are three date values (”time”) which are added in ”NewDB”, The steps to verify that the value are:

```
has (y ) Add  "2018-04-28" (
    Add ( "2018-04-27", Add ( "2018-04-26", NewDB )
```

```
))

E2 = ("2018-04-28" = = y) v has y (
    Add ( "2018-04-27", Add ( "2018-04-26", NewDB )
))

E2 = ("2018-04-27" = = y) v has y (
    Add ( "2018-04-26", NewDB )
))

E2 = ("2018-04-27" = = y) v has y (NewDB)

-> True
```

A natural transformation corresponds to a relation between two functors. As an illustration, the graphs mentioned above can be represented in a different way by considering the functor $G'\ (N) = P\ \ (N\ \times P\ \ (N \times N))$ that associates adjacent links to each node. The relation between G and G' can then be represented by a natural transformation $\eta\ :\ G'(N)\ \rightarrow\ G(N)$. This one can be defined, by using set comprehension notation, as: $\eta(g') = (x, y, z)\ |\ (x, ys) \in g, (y, z) \in ys$. This transformation is invertible and the functors/datatypes are then said to be naturally isomorphic $G'(N) \cong_\eta G(N)$. If the two structures represent a "same" information, the performance of a program depends on the structure selected. As an example, a function/program to get the adjacent links, i.e. $get(n) : G(N) \rightarrow P(N \times N)$, will have a complexity $O(n)$ where n is the number of edges when using G, and $O(m)$ where m is the number of nodes when using G', and $m \leq n$. So, $get'(n) : G'(N) \rightarrow P(N \times N)$ is "faster" than $get(n)$. The change from G to G' can be viewed as an optimization technique called "memorization" in the sense that G0 memorizes the result (i.e. adjacent links) for each input node and then eliminates extra computations [37]. The optimized version of the program will be obtained with $get'(n) = get(n) \circ \eta - 1$ that can be simplified by using the definitions of g and $\eta - 1$ (and is known as short-cut fusion optimization [34]). Another common

optimization technique consists in splitting data and using parallel computations. In the example of graphs and by considering a pair of computers, this can be modeled with $G''(N) = G(N) \times G(N)$. The function to get the adjacent links will be now $get''(n)(g1, g2) = \cup \circ get(n)(g1) \times get(n)(g2))$with a complexity $O(max(n1, n2))$ where ni is the size of gi. And finally, we get an optimization chain that can be represented by: $O(get'') \leq O(get') \leq O(get)$.

2.4.2 Data models

Data should not be stored in a machine in a chaotic manner. Otherwise, not only the logical relationship between the information will be lost, but also the searching efficiency will also be relatively low. Therefore, it is necessary to discover some inter-relationships between these data and use them to organize the data into a logical structure [48].

List model

List [49] [50] is a finite sequence which is composed of some elements or empty elements. List could also be defined with a mathematical model. List can be expressed as $(a_1, a_2, a_{i-1}, a_i, a_{i+1}, ..., a_n)$, where a_{i-1} is ahead of a_i, a_i is ahead of a_{i+1}, a_{i-1} is said to be the direct precursor of a_i, and a_{i+1} is the direct successor of a_i. Figure 2.39 describes this list. However, a data element can consist of several data items in some complex lists.

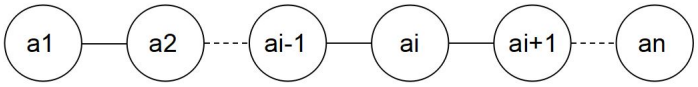


Figure 2.39: Example of a simple list

Figure 2.40 is a complex list, the three elements (time, latitude, longitude, depth, mag, Source) are data items

When $i = 1, 2, 3...x - 1$, a_i has one and only one direct successor. When $i = 2, 3..., n$, a_i has one and only one direct precursor. Therefore, the number n of the list elements is defined as the length of the list. Empty list is composed

time	latitude	longitude	depth	mag	Source
2018-04-28	33.4893333	-116.7948333	10	4.8	us
2018-04-27	18.0433	-67.4248	114	2.74	pr
2018-04-26	-6.1569	143.047	5.311	2.9	tul
2018-04-25	-21.0123	-178.7933	13.98	0.95	ci

Figure 2.40: Example of a complex list

by ($n = 0$) list. However, each element in a non-empty table has a certain position. For example, a_1 is the first data element, a_n is the last data element, a_i is the i-th data element, and i is the bit order of the data element a_i in the list. Therefore, a list is a sequence firstly. In other words, there is an order between elements. Then, list emphasis is limited. However, when the user needs to insert and delete elements in a List, the user needs to move a lot of elements. The time execution depends on the number of elements in this list.

Data types used in computer science are generally composed with basic data types (booleans, integers, characters, etc.), structured data types (structures and union types) and collections (e.g. arrays or recursive datatypes such as linked lists or binary trees). If most of the imperative programming languages (e.g. C) mainly use arrays and loops (for or while statements), functional programming languages use recursive datatypes (with in particular recursive lists) and recursive functions. As mentioned previously, lists can be modeled both by a parameterized data structure $L(x)$ where x is a type parameter restricting the type of the elements contained in the list, and two "constructor" functions [34] to create a new empty list $e : 1 \rightarrow L(x)$ and to add an element in a list $a : x \times L(x) \rightarrow L(x)$. In the preceding definitions, 1 can be interpreted as the "void" element found in imperative languages, and $x \times L(x)$ as a pair of values. The function e then returns a constant value and is then simply written $e : L(x)$. As an illustration, the list $xs = [x1, x2, x3]$ will be constructed with $a(x1, a(x2, a(x3, e)))$. As a complement, it is well known by functional programmers that a binary function is equivalent to a higher-order function, i.e. $x \times y \rightarrow z \equiv x \rightarrow (y \rightarrow z)$, and the preceding definition of the list xs is also

equivalent to a x1 (a x2 (a x3 e)).

Most of the functions on lists $f : L(x) \rightarrow y$ can be defined by case analysis and two equations: $fe = v$ and $f(axxs) = gxy$ where $y = fxs$. Such a function is called a "catamorphism" [62] and can be written $f = [v, g]$. As sample uses, the function returning the length of a list can be simply defined by length $xs = [0, \lambda x.\lambda y.1 + y](xs)$ where $\lambda x.\lambda y.1 + y$ denotes an "anonymous" function $hxy = 1 + y$, the concatenation of two lists defined by $xs \oplus ys = [ys, \lambda x.\lambda y.axy](xs)$, the application of a function h to all the elements of a list by $maphxs = [e, \lambda x.\lambda y.a(hx)y](xs)$, the test of the presence of an element z in a list xs by contain $xsz = [False, \lambda x.\lambda y.(x = z) \vee y](xs)$, the selection of the elements satisfying a predicate $p : x \rightarrow Bool$ by $filter\ p\ xs = [e, \lambda x.\lambda y.if\ (p\ x)\ then\ y\ else\ (a\ x\ y)]\ (xs)$, etc. This latter function is the key point to search for specific information in a list. For instance, the search of the common elements on two lists can be obtained with $xs \cap ys = filter\ (\lambda x.contain\ ys\ x)\ (xs)$. As a remark, queries (with filter) are often associated with transformations (with map) with the list "comprehension" [67] notation: $fx \mid x \in xs, px \equiv map\ f\ (filter\ p\ xs)$. The code below presents a sample implementation of the preceding functions in Haskell with their respective complexity. Lists are already present in the language, and $L(x) \equiv [x]$, $e \equiv []$ and $(a\ x\ xs) \equiv (x : xs)$. In the code, the last function simply removes duplicate elements in a list which is also a transformation of a list into a set (if adding the extra assumption that the order of the elements is not important).

```
cata v f [] = v
cata v f (x:xs) = let y=cata v f xs in f x y
length xs = cata 0 (\x -> \y ->1+y) xs
-- O(n) , n=length xs
xs ++ ys = cata ys (\x -> \y ->x:y) xs
-- O(n)
map h xs = cata [] (\x -> \y ->(h x):y) xs
-- O(n)
contain xs z = cata False (\x -> \y ->(x==z)||y) xs
-- O(n)
```

```
filter p xs = cata [] (\x -> \y ->if (p x) then
x:y else y) xs
-- O(n) xs /\ ys = filter (\x -> contain ys x) xs
-- O(n*m), n=length xs
--m=length ys
set xs = cata [] (\x -> \y ->if contain y x then y
else x:y) xs -- O(n^2)
```

We must notice that the preceding functions satisfy certain properties, such as $filter\ p\ (xs \oplus ys) \equiv (filter\ p\ xs) \oplus (filter\ p\ ys)$ or $filter\ p\ (filter\ q\ xs) \equiv filter\ (\lambda x.(px) \wedge (qx))\ xs$, that can be used to improve the performances of the functional programs. In particular, the first property is interesting with concurrency (i.e. one program searching in xs and the other one in ys), and the second to avoid a loop (i.e. testing p and q in the same loop). If the oblivion of the ordering relation is used to pass from lists to sets, the addition of an ordering relation between the elements contained in a list can be used to improve the search of a particular element in a faster way, e.g. by stopping the test of the presence of an element x when finding some i such as $x < xi$. The code below then gives the example of the insertion sort to transform a list in an ordered one (with an ordering relation $o : x \to x \to Bool$). It also gives the new version of the function "contain xs z" with the ordering operator o as an extra parameter.

```
insert o y [] = [y] -- O(n)
insert o y (x:xs) =
           if (o y x) then (y:x:xs)
           else
           x:(insert o y xs) sort o xs
           = cata [] (\x -> \y ->insert o x y) xs
-- O(n^2)

contain' o [] z = False
-- O(n) statically
contain' o (x:xs) z =
              if (x==z) then True
              else if (o z x) then False
```

```
                else (contain' o xs z)
```

The (natural) transformation to switch from a list $L(x)$ to an ordered list $OL(x)$ is not invertible in the sense the position of the original elements are forgotten. This can be changed by embedding explicitly the location of each element to an intermediate data structure, i.e. $[x1, ...xn] \mapsto [(x1, 1), ...(xn, n)]$. This transformation can be achieved by using a function zip such as $sip[x1, ...xn]$ $[y1, ...ym] = [(x1, y1), ...(xk, yk)]$ where $k = min(n, m)$. Thus, by shifting the ordering relation o to $O'(x, y)(x', y') = o\ x\ x', sort'\ xs = sort\ o'(zip\ xs[1..(length\ xs)])$ with return an ordered list with the original position of the elements. For instance, $sort'(<)[5, 1, 8, 12, 5, 8]$ be equal to $xs' = [(1, 2), (5, 5),$ $(5, 1), (8, 6), (8, 3), (12, 4)]$. Now, it is possible to find again the unordered list by sorting the position. The code below gives the implementation of all these functions, and establishes the natural isomorphism $L(x) \equiv_{sort'} OL(x)$.

```
zip [] _         = [] zip _ [] = []
zip (x:xs) (y:ys) = (x,y):(zip xs ys)

sort' o xs =
    sort (\x -> \x' -> o (fst x) (fst x'))
(zip xs [1..]) sort1' oxs =
    sort (\x -> \x' -> (snd x) < (snd x')) oxs

contain2 o z [] =
          False contain2 o z (x:xs) =
          if ((fst x) == z)
          then True else
          if (o z (fst x))
          then False else (contain2 o z xs)
```

As a complement, it is also possible to establish an isomorphism with sets $S(x)$, i.e. $OL(x) \equiv_{set'} S(x)$, by considering the following code. In particular, $set'\ xs'$ will be equal to $[(1, [2]), (5, [5, 1]), (8, [6, 3]), (12, [4])]$.

```
set' []  = [ ]
```

```
set' [x] = [x]
set' ((x,n):((x',n'):xs)) =
             if (x==x')
             then set' ((x,n++n'):xs)
             else (x,n):(set' ((x',n'):xs))
set1' [] = [ ]
set1' ((x,[n]):xs) =
            (x,n):(set1' xs) set1' ((x,(n:ns)):xs)
                    =
            (x,n):(set1' ((x,ns):xs))
```

So, there are three equivalent representations of a single item of information and three equivalent queries with various performances: $contain : L(x) \rightarrow Bool$, $contain2 : OL(x) \rightarrow Bool$ and $contain2 : S(x) \rightarrow Bool$. As a sample concrete use of these elements, it is possible to consider a word search inside document with for instance the "Alice's adventures in wonderland (Lewis Carroll)" book [6]. This document has an original file size of 154 ko and is composed of a list l of 27341 words - which corresponds both to the length of $l : L$ and $l' = sort'\ l : OL$, but only 2955 distinct words - what correspond to the length of $l'' = set'\ l' : S$. The search for the word "childhood", that appears near the end of the document, with these various values leads to the measures presented in Figure 2.41. This one also presents the file sizes used to store l, l' and l''. Indeed, the original document corresponds to a list of characters that has to be split into a list of words and the transformation requires a certain amount of time (Figure 2.42), and storing this list of words avoids these extra computations. This is similar for the sorting of a list or its transformation to a set of words, and the Figure 2.42 well show the algorithmic complexity: 0,322s for set' that id O(n) and 36,933s for sort that is O(n2). Finally, Figure 2.43 summarizes the main elements presented in this part.

It presents a sample list of integers (but other types of elements are possible such as words, for instance). It gives the invertible natural transformations used to establish "equivalence" and the various functions to query a specific infor-

[6] https://www.gutenberg.org/ebooks/11

model	**t**ime (s)	**d**b size (ko)
L	0.009	190
OL	0.006	397
S	0.005	191
T	0.004	209

Figure 2.41: Measurements.

transformation	**t**ime (s)
$words : 1 \to L$	0.021
$sort' : L \to OL$	36.933
$sort' : OL \to S$	0.322
$tree : S \to T$	0.123

Figure 2.42: Measurements(cont.)

mation in each representation. Finally, it gives the performance of these functions and shows how data transformations can lead to an optimization chain (T being the limit of ($\geq$) in the figure).

Table model

Table model [51] is a basic data model (see in Figure 2.44). Table model is a sequence of the same type of data arranged in a certain order. Therefore, the data model could be defined by a "Field" (see in Figure 2.45). On the other hand, there is also a table in this model. The "record" can be written as: (a1, ..., an). The sequential storage structure of List is also the same and finds space in memory, and takes up a certain amount of memory space by placeholders. Then, the data elements are stored sequentially in this open space. That is, the first data element is stored in the array marked 0. Figure 2.46 presents a Maximum size of a list.

In Figure 2.45, Database $< x >$ is a parameterized data type. This type defines the operation to construct a new database by "$newDB()$". "$add(Records)$"

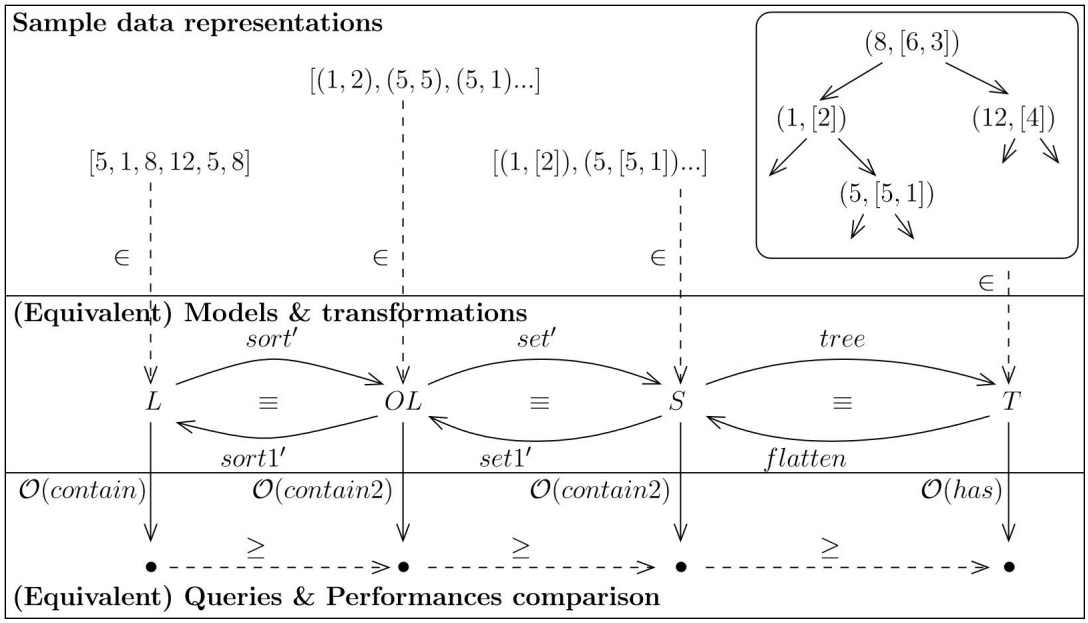


Figure 2.43: Summary.

could be used to add a new value from "Records". Verifying the value in this database by using"$has()$" and getting some adapted values " $get()$ " to construct this new database. Data type "Record" describes all of the data type which it will be used in this table. To model Record, a product can be used as the fields are all known at action time. The table is like a collection and should be modeled by a list functor. More generally, these elements can be represented by using the graphical notation as follows (Figure 2.47).

time	latitude	longitude	depth	mag	Source
2018-04-28	33.4893333	-116.7948333	10	4.8	us
2018-04-27	18.0433	-67.4248	114	2.74	pr
2018-04-26	-6.1569	143.047	5.311	2.9	tul
2018-04-25	-21.0123	-178.7933	13.98	0.95	ci

Figure 2.44: Example of table in a relational database

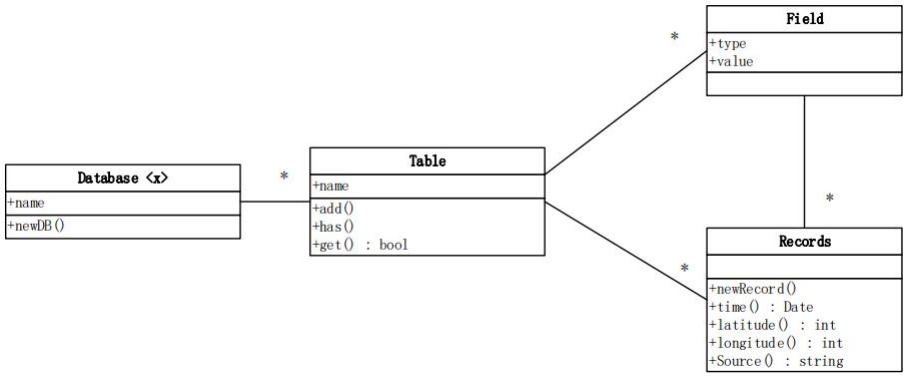


Figure 2.45: Example of Database schema

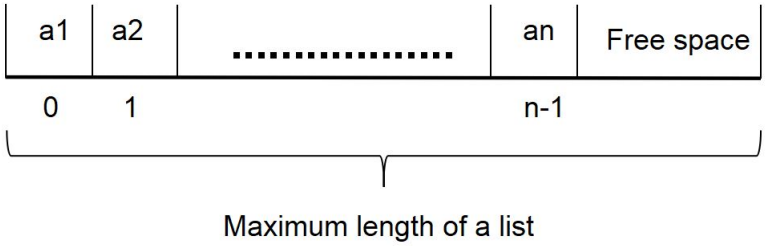


Figure 2.46: Size of a list

The parameterized data type DB(X) will correspond to a functor by defining (latter) an extra morphism $DB(f) : DB(X) \to DB(Y)$, where $f : X \to Y$. This morphism will represent, for instance, the application of "f" to all the elements of the database (by preserving the structure). Eg. DB(latitude)(db) will return the db contain only the latitudes of all the records in $db : DB(Records)$.

For this model, we could use Product data types. Product data types x $\times$ y that are similar to records (i.e. a finite set of fields having eventually different types) can be used with lists to define tables or relations $R(x, y) \equiv L(x \times y)$ as found in relational databases. We recall that binary products are specified with two functions: $fst : x \times y \to x$ and $snd : x \times y \to y$. Thus, all the functions on lists can be used on tables, with a particular "filter" to query specific information. The code below gives an example of a table and a sample query

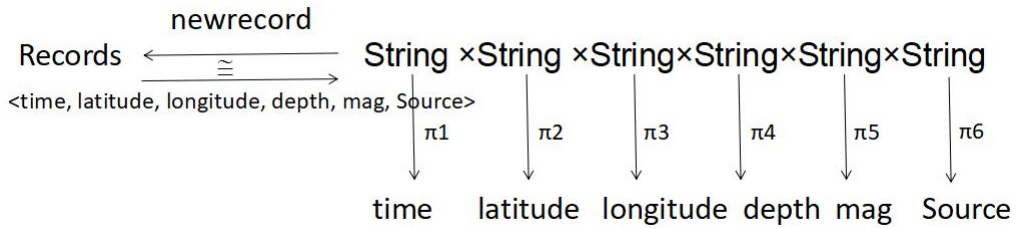


Figure 2.47: Example of table schema

to get the ”Time” (first field) of the earthquake has happened in ”us” (second field) with its equivalent ”comprehension” representation. It also shows how easy it is to generalize the example to get other values with the is x function (e.g. query1 == is ”us”), and how to combine it with the function presented in the preceding part to get more complex queries (see query2).

```
type Relation x y = [String]
type Time = String
type Source = String
type latitude = String
type longitude = String
type depth = String
type mag = String
table1 :: Relation Name Job table1 = [
   ( "2018-04-28", "us",
     "33.4893333", "116.7948333", "10", "4.8"
   ),
   ( "2018-04-27", "pr",
     "18.0433", "-67.4248", "114", "2.74"
   ),
   ]
query1 = [ fst r | r<-table1, snd r == "us"]
query1 =
map fst (filter (\r -> (snd r) == "us")table1)
-- O(n)
   is x = map fst (filter (\r -> (snd r) == x)table1)
-- O(n)
   query2 = (is "us") /\ (is "pr")
-- O(n^2)
```

Relations are associated to specific operators such as inversion of a relation, to get the image of an element, and the join of two relations that can can defined as follow:

```
inverse :: Relation x y -> Relation y x
inverse xs = map (\r -> (snd r,fst r)) xs

img :: Relation x y -> x -> [y]
img xs z = map snd (filter (\r -> (fst r) == z) xs)

join :: Relation x y -> Relation y z -> Relation x z
join xs ys ...
```

Thus, is $x \equiv img\ \ (inverse\ \ table1)\ \ x$

Map model

Each element of a map [51] [52] is called a key-value pair. Map also has its own rules when looking for elements in Map. Map retrieves the corresponding value (value) by looking for the key (key), and the value of key cannot be duplicated. Map models are defined by a set of values which is specified by a schema in Figure 2.48. As a list is stored, each object in this database, includes the association of objects with a key K, their corresponding value type T and reference type R. In Figure 2.49, the data of time is key, and the other data are values. Therefore, the structure of objects is represented by $o = (i, (K, T, v), (K, R, ref))$. There is an example in Figure 2.49. More generally, Map model corresponds to $DB(Z) : DB(E) \rightarrow DB(F)$ where $Z : K \rightarrow V$. K is a set of possible keys and V is a set of possible values. Representation of a list as a Cartesian products such as: $V = v_1 \times v_2 \times v_3 \times ...v_k$. Figure 2.50 describes a schema to represent "$get()$ with K.

For instance,

```
size :: Map k a -> Int
```

```
size empty                                          == 0
size (singleton 1 'a')                              == 1
size (fromList([(1,'a'), (2,'c'), (3,'b')]))  == 3

--- O(1). The number of elements in the map.
```

size empty == 0 size (singleton 1 'a') == 1 size (fromList([(1,'a'), (2,'c'), (3,'b')])) == 3

— O(1). The number of elements in the map.

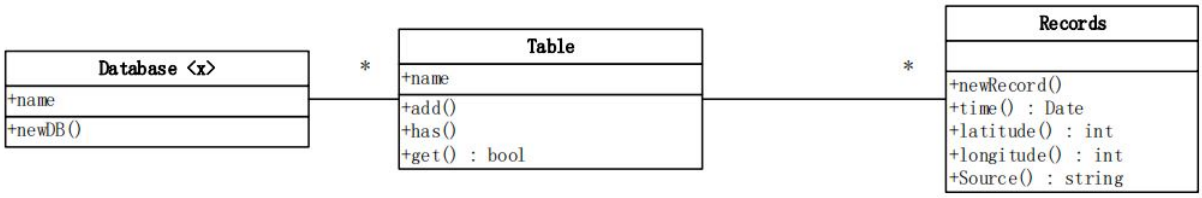


Figure 2.48: Key value and Document database schema by UML

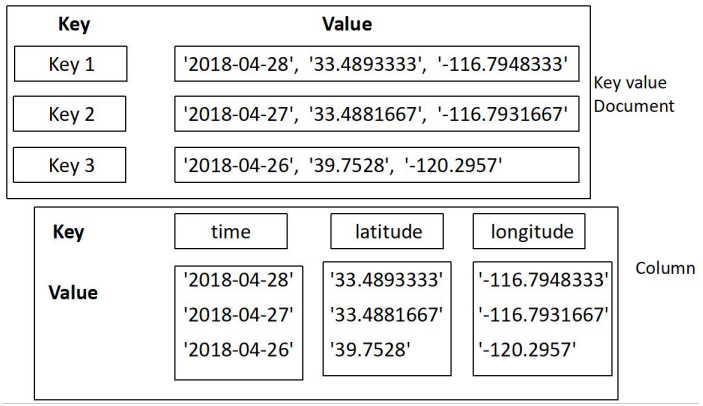


Figure 2.49: Key value, Document database and Column database

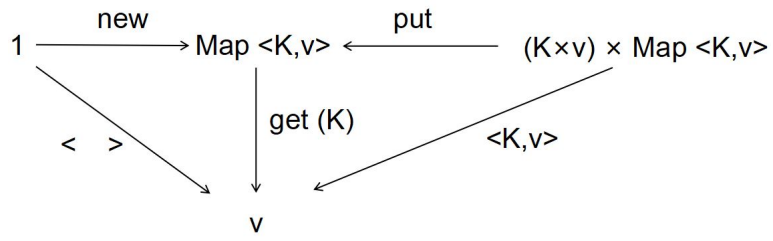


Figure 2.50: Schema for Map model

Graph model

In the previous three mathematical models, there is a simple one-to-one, one-to-many relationship between data elements. However, if there is a more complex relationship between data elements, Graph is a more effective method. For example, the earthquake information. In the period after the earthquake occurred at Santiago(USA), there were several aftershocks in the periphery. In the analysis of data, we can use Graph to establish the connection between the different earthquakes which occurred in the same region.

In general, a graph [51] [52] can be represented by a set of edges and a functor $G(N) = \mathcal{P}(N \times N \times N)$ where N represents a set of nodes and $N \times N \times N$ edges of the form (source, label, destination). With a function $f : N \to N'$, we can define a graph morphism $m(f) = \mathcal{P}(f \times f \times f)$ that changes the nodes by preserving the structure of the graph - i.e. if (x, y, z) is an edge of g then $(f(x), f(y), f(z))$ is an edge of $m(g)$. $m(id_N)$ is an identity morphism, morphisms are composable (i.e. $m(f \circ g) = m(f) \circ m(g)$) what makes the set of graphs and morphisms another example of a category called $\mathcal{G}raph$. Figure 2.51 describes an example of a graph.

2.4.3 Queries

Introduction

A collection generated by all the data elements of the same type that needs to be collected is called a Searching Table. A key is a value of a data item in a data element, which can also be called a key-value. Key could be used to identify a

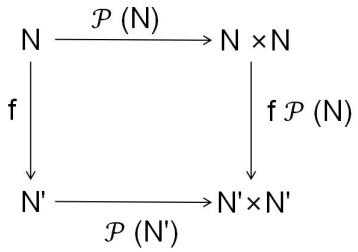


Figure 2.51: Example of a graph

data element or to identify a data item (field) of a record. If this keyword (key) can uniquely identify a record, then this keyword is called the primary key. Hence, Queries determine a data element (or record) whose keyword is equal to the given value in a Searching Table according to a given value. Figure 2.52 describes an example of searching table.

collection of data elements

Key value

time	latitude	longitude	depth	mag	Source
2018-04-28	33.4893333	-116.7948333	10	4.8	us
2018-04-27	18.0433	-67.4248	114	2.74	pr
2018-04-26	-6.1569	143.047	5.311	2.9	tul
2018-04-25	-21.0123	-178.7933	13.98	0.95	ci

Figure 2.52: Example of searching table

In general, there are two types of Searching Tables: Static Search Table and Dynamic Search Table. The Static Search Table is a searching table that is only used for lookup operations. Its main operations are:

1) Querying whether a particular data element is in the searching table

2) Checking a specific data element and various attributes

Therefore, the Static Search Table is used to query the data existing in the database.

Dynamic Search Table: Inserting a new data element that does not exist

in the searching table with the lookup process, or deletes a data element that already exists from the searching table. Therefore, the two modes of the Dynamic Search Table are:

1) Inserting data elements

2) Deleting data elements

In order to improve the searching efficiency, we need to organize the data structure specifically for the lookup operation. This data structure for the lookup operation is called the querying model. In terms of logical structure, querying model is based on the collection. There is no essential relationship between the records in the collection. In order to achieve higher searching performance, we have to change the relationship between data elements. In this chapter, we introduced models for several queries.

Table model

With the definition of table model, the query model has been established below (see in Figure 2.53):

In this figure, the query is presented by $X \xrightarrow{\text{get(Y)}} Y$. When the record is empty, the answer will also be returned by an empty value. The query has been used to get the result. The mathematical expressions are:

$get(Y) : Y \times [Y \times X] \rightarrow [Y]$

$get(Y, DB) = concat \circ map\ (\backslash(x, z).(x = Y)(DB) = [\ Y|(x, z) \in DB,$

$x = Y\ ]$

If the result happens to be in the first position of the first column or in the first position of the first field, the value of time complexity is O(1). Usually, time complexity of this model is O(n).

Lists model

As the structure of the lists' model, the query schema is similar with the query in the table model. However, the storage unit is a List. The time complexity depends on the number of lists and the number of items in the list ($O(n)$). However, if the user only searches the first column value in this database, the

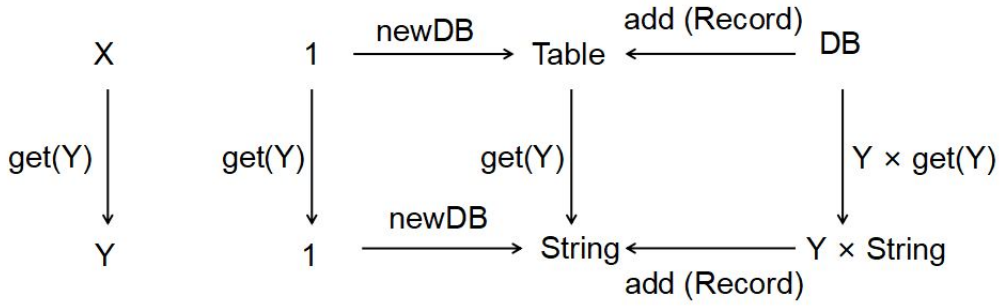


Figure 2.53: queries for table model

time will be short. Because the elements are stored in the array marked 0 (the first element). On the contrary, if the query needed to obtain all the information with a special data, the time to work is also long. Figure 2.54 presents an example for querying a list model. The mathematical expressions are:

1) $get(Y) : Y \times [Y \in DB] \rightarrow [Y]$

2) $get(Y, DB) = concat \circ map\ (\backslash(x, z).(x = Y)(DB) = [Y|(x, z) \in DB, x = Y \wedge (DB)]$

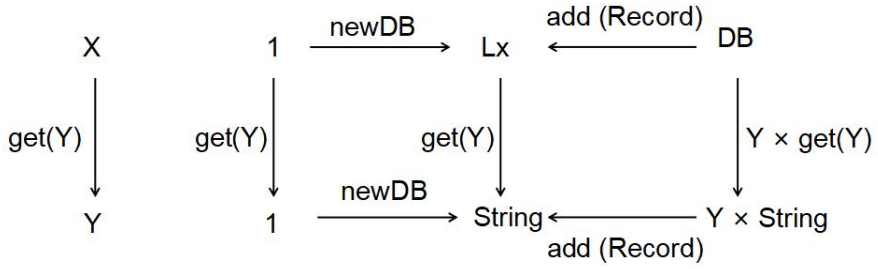


Figure 2.54: queries for list model

Map model

Queries could be described by using the tuple of list (see in Figure 2.55). This example relies on the query: if the value is in this database? When the DB is empty, the result is always ”False”. On the contrary, if there are values in this list, the value will be compared to the first value (key) in this list, then,

comparing the value with the rest of this list (xs). The time complexity of the query is in O(n).

put(Object key, Object) There are two parameters in this method, one is "key" and the other is "value". In the newly added element, it needs to specify the key and value of the new element, and can only add elements to the end of the array.

get(Object key): Map is to find the value through key, the "get" method needed to pass in the key value.

The mathematical expressions (get) are:

1) $get(Y) : Y \times [[\ ] \in DB] \rightarrow [\ ]$

2) $get(Y, DB) = concat \circ map(\backslash(x, z).(x = Y)(DB) = [Y|(x, z) \in DB, x = Y \wedge (DB)]$

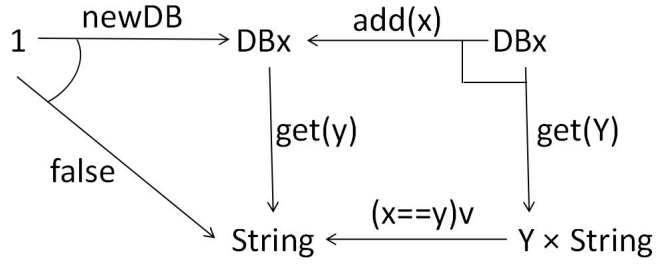


Figure 2.55: Example of queries

Graph model

Querying a particular information in a graph model is also a way to find a morphism from the graph representing (a query) to a graph database. Indeed, a query such as "(X is-Source US and (X has-latitude Y)" can be viewed as a labeled graph with two edges $\{Source : X \rightarrow US, latitude : X \rightarrow Y\}$, where X and Y denote variables as illustrated in Figure 2.56. The query is a graph and an element of $G(N \cup X)$. The result is then a set of pairs $\{\{(X, US), (Y, 33.489333)\}\}$ and the program finding the possible morphisms can be formalized by a function $unify : G(N') \times G(N) \rightarrow \mathcal{P}(\mathcal{P}(X \times N))$ if

$N' = N \cup X$ the union of constant nodes N plus variables X. In general, the time complexity of this model with n node and e edges is 0 (n+e).

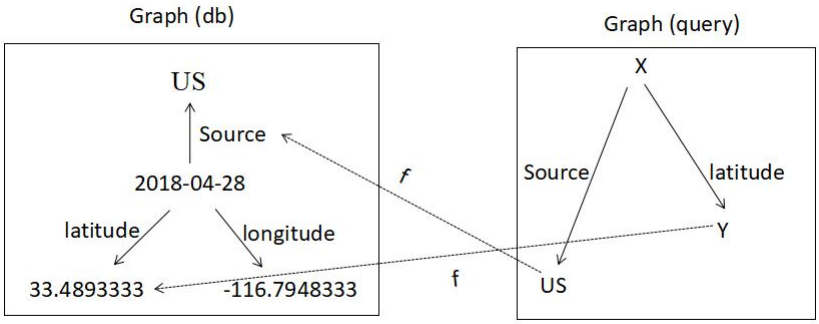


Figure 2.56: Sample graphs and query/morphism.

Propositions

- Strategy
- Process followed in our study

Chapter 3

Propositions

3.1 Strategy

In the chapter "State of art", we have seen that there are many databases with different structures. Nevertheless, through our research, we will assume that at a high level they rely on only three main data structures :

- Table modeling by list of tuples / records: $[(x, y)]$;
- Map modeling by an indexed tuple $[x, (x, y)]$;
- Graph modeling by a collection of triple $[(x, p, v)]$ could be indexed or not;

The structure of list of tuples models a sequential searching policy in an unorganized dataset. This is the model which could be used when the query haven't the skill to be preprocessed. Hence, the query processor needs to scan all the entries to select the relevant ones.

The map structure models a searching standard that benefits from precomputed ordered indexing. This indexing makes it easier to select, from a simple arithmetic calculation the first element satisfying the condition and then a simple path to select subsequent ones.

The graph structure models a searching policy that selects subgraphs from a basic pattern. This approach assumes that the data is organized according to a suitable schema that ensures rapid selection.

All of these structures can be viewed as morphism which could be set up as dedicated partitions from an anonymous set. We consider here that we use ”set” as our main category.

Indeed, if we observe the behavior of a relational database when it has to perform a query, either the query is on the main key and the table behaves like a map or the query is about not indexed data, that is, the query is used for a datum in the tuple and then the database is forced to scan the contents of the table as a query in a table.

In the case of a document-oriented database, there is a behavior which is similar to the relational database: either the query concerns the id of the table in which case the search is similar to a search in a map, otherwise, this is a sequential search. The document-oriented query processor is similar as in relational database, except that it is possible to create many indexes on several different data in the tuple and in this case, the processor of the request will have to use a policy based on the pre-calculated tuple map.

Key value database seen with a similar approach.

For a column-oriented database, it is a view of several maps of cascaded tuple done skillfully by the table's designer. Thus, a query that is correctly aligned with the index structure of the table will actually make successive selections based on the map policy, and if the query leaves the indexing strategy then the table policy will be used.

For a column-oriented database, if we set the column as an index which we want to query, the rate of the query will be greatly improved. On the other hand, there will be no major changes to the modification of this type of database.

Finally, the graph base, the graph solution will obviously be the most suitable and the selection will be sure to make the pattern matching to the reformulated query in the form of a subgraph.

3.2 Process followed in our study

Our study has concentrated on reading a reference document and putting it in a pivot form that should be a list of tuples (n-areas in the case of tables and triples in the case of graphs). In Haskell, it is easier to have a table. When the table is loaded in memory, we apply a request and measure the processing time. We will obtain a processing time comparable to that of the same operation on a database when the query does not concern the index or the indexes available. Then a functor will allow us to transform this table to a map of tuples. The request will be accordingly transformed (using the same Functor) to perform the same operation but on the new structure. The measurement of the execution time will make it possible to obtain the processing time for a query where the index or the indexes can be pre-computed. Another Functor will make it possible to transform the table of tuples towards the structure of graph of triples. The transformation of the request into a selection pattern adapted to this new structure will make it possible later to measure the processing time.

To ensure a comparison with the products of the trade, we will also propose some morphisms which will allow us to transform the different representations into a concrete representation belonging to standard databases (MongoDB, Neo4J etc) and the queries are transformed too.

3.2.1 Procedure for importing initial data

We have to select a dataset to perform our tests. It turns out that it's in the form of a CSV file which contains a table with X rows and Y columns.

An interpreter for CSV can be defined in Haskell and by considering the simplified grammar:

```
<val> := 'a' | ... | 'Z' | '0' | ... | '9'
<csv> := (<val>*(,<val>*)*'\n')*
```

From the grammar, the code for the parser can be derived as below. The notation `[x..y]` represents the enumeration form `x` to `y`, `++` is the list concatenation operator and the `do` notation is used for sequential composition. The

”oneOf” function then returns a parser that checks if the first element of an input text belongs to the parameter of the function. The ”many” function represents the repetition of an element and corresponds to the * operator in the EBNF expression.

```
val :: Parser Char
val = oneOf (['a'..'z']++['0'..'9']++...)

csv = many $
do many val
many $ do oneOf [',']
many val
oneOf ['\n']
```

The program above is then extended to extract some information as follows. The `x<-` notation simply introduces a new variable `v` that stores an information, and `return` specifies the result of the function. The `(:)` is, as mentioned before, the Haskell operator to add an element to a list.

```
csv = many $
do v  <- many val
vs <- many $ do oneOf [',']
many val
oneOf ['\n']
return (v:vs)
```

Next, the ”fromCSV” function uses this parser to read a csv file with and get a two-dimensional array of strings (`[[String]]` in Haskell).

```
fromCSV file = do
str <- readFile file
let Right dta = parse csv "" str
return dta
```

The ”toCSV” function allows us to write back in a CSV format on array of String.

```
toCSV file dta = do
let lines = map (intercalate ",") dta
let str   = intercalate "\n" lines
writeFile file (str++"\n")
```

3.2.2 Functional descriptions by table

Definition

Table model is a simple manner to organize a dataset is in using table as illustrated in the Figure 3.1. Such a table could be modeled by a list of lists, and the functor $X = [[String]]$.

Row key	time	latitude	longitude	depth	mag	Source
usgs1	2018-04-28	33.4893333	-116.7948333	10	4.8	us
usgs2	2018-04-27	18.0433	-67.4248	114	2.74	pr
usgs3	2018-04-26	-6.1569	143.047	5.311	2.9	tul
usgs4	2018-04-25	-21.0123	-178.7933	13.98	0.95	ci

Figure 3.1: Sample table

Morphism

As the definition of table model, we could define two morphisms. The first one which we have used is $get()$. This morphism is used to search that relevant data in a data set. The mathematical model is : [[String]] $\overrightarrow{get}$ [String]. However, if there is a set of results in this data set, the morphism has changed to: [[String]] $\overrightarrow{find}$ [[String]]. The schema model to query the values in a table is in Figure 3.2.

There is an example below:

- $get_{String}[[String]] = \{String \in [[\ ]] \mid \exists\ [[x, String]], x = String\}$

For this equation, it has been shown that if there is a suitable list of $String$ in the last column of the list of $String$, the result could be obtained a list composed by list of $String$. Such a table could be modeled by a list of tuples or a list of list which uses a structure like $F_1[[String]]$.

Thus, the dataset of the Figure can be encoded in Haskell as follows:

```
type X = [[String]]

D :: X
D = [["2018-04-28", " 33.4893333",  "-116.79483",
     "10", "4.8", "us"]
,["2018-04-27","18.0433", "-67.4248", "110", "3.23", "pr"]
,["2018-04-26","-6.1569", "143.047", "47", "3.46", "pr"]
,["2018-04-25","-21.0123","-178.7933", "111.1", "1.3", "ak"]
]
```

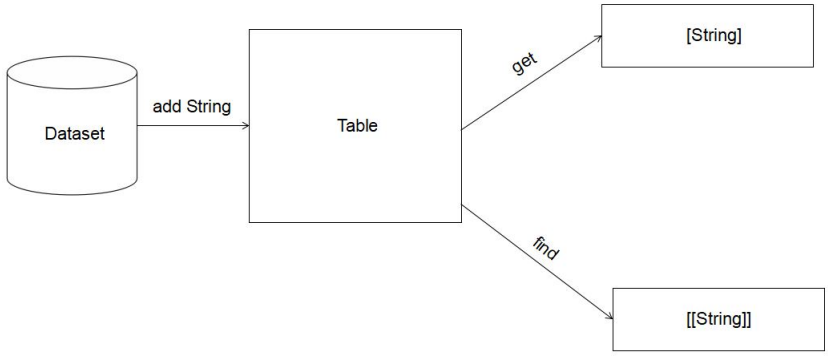


Figure 3.2: Schema model by table.

By using this representation, queries can be easily expressed by using list comprehensions. As an illustration, the code below could be used to get the full tuple such as $"Source" = "US"$, which is similar to the SQL statement: $SELECT * FROM Y WHERE "Source" == "US"$.

```
query :: X -> [String]
query D = [o| [time, latitude, longitude, Source]
<- (tail D), "Source" == "US"]
```

3.2.3 Functional descriptions by Map

Definition

Operators on the set in the previous part are then defined by using functions and rewriting rules such as: $e_1 : p(z, m) = z$ and $e_2 : p(s(n), m) = s(p(n, m))$. These rules correspond to the addition, and one can check that $p(s^2(z), s^1(z)) =$

$s^3(z)$ by applying e_2 twice then e_1. Now, the expressions defining N can formalized by the way of a grammar:

```
<N> := z | s(<N>)
```

Or by using a datatype definition in any (functional) programming language. The code below then gives an implementation of (N, p) in the Haskell programming language.

```
data N = Z | S(N)

p(Z,m)    = m
p(S(n),m) = S(p(n,m))
```

As another example, lists of N can be specified by a constant for the empty list $e : L_N$, and a function to add a value to a list $a : N \times L_N \rightarrow L_N$. Thus, an expression such as $a(n_1, a(n_2, e))$ will be interpreted as a list $[n_1, n_2]$. The catenation operator is then defined in a similar manner of the plus operator on numbers with: $p_1 : pl(e, l') = l'$ and $p_2 : pl(a(x, l), l') = a(x, pl(l, l'))$. One can then check for instance that $pl([1, 2], [3]) = [1, 2, 3]$ by applying p_2 twice then p_1.

Lists can be generalized by using a parameterized datatype $L(x)$ where x is a type variable replacing N in the previous definition (and thus $L_N = L(N)$). Some generic functions, used in the rest of the document, can then be defined to: concatenate a list of lists (cat), apply a function to all the elements of a list ($map(f)$), select the elements satisfying a predicate ($filter(p)$), etc. The implementation of these functions in Haskell are given below.

```
data L x = E | A (x,L x)

pl(E     ,l') = l'
pl(A(x,l),l') = A(x,r)
where r = pl(l,l')

cat(E)      = E
cat(A(x,l)) = pl(x,r)
where r = cat(l)
```

```
map(f,E)       = E
map(f,A(x,l)) = A(f(x),r)
where r = map(f,l)

filter(p,l) = r'
where f(x) = if (p(x)) then A(x,E) else E
r  = map(f,l)
r' = cat(r)
```

Morphisms

The morphism for Map model is: Y = [String, [String]] $\overrightarrow{get}$ [String]. This is used to search only one relevant data in this list. This model [String, [String]] $\overrightarrow{find}$ [[String]] can be used to search a set of results in this data set.

Functor

Two functors (Figure 3.3 and Figure 3.4) can be applied to change the Table model to a Map model.

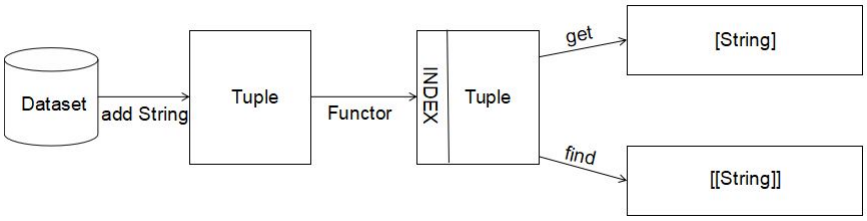


Figure 3.3: Sample functor 1.

We could firstly add the data into a table model. Then, the indexed table could be obtained by a functor. The "only one" or "a set of data list" could be acquired by the morphism get and $find$.

The second functor is similar to the first functor. The difference is that we first directly changes the original dataset to the index dataset and then add it to the table.

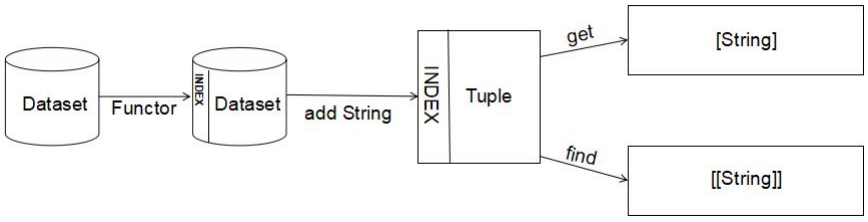


Figure 3.4: Sample functor 2.

3.2.4 Functional descriptions by Graph

Definition

Graph model corresponds to a new functor Z = [(String, String, String)]. Where the first String is the entry, the second one is the property name and the last one is value. For example, in Figure 3.1, we can model the first row as an entry N_1 with a relation r_1 (about time) to value E_1 (eg., 2018-04-28), relation r_2(latitude) to value E_2 (33.4893333), relation r_3 (longitude) to value E_3 (-116.79483) and the last relation r_4 (Source) to value E_4 (US). This model is described in Figure 3.5. Therefore, Graph databases use labeled graphs that can be represented by $Z(X) = L(X \times (X \times X))$ where the first X corresponds to the source of an edge, the second to the label and the third to the destination.

Morphisms

The two morphism for Map model is: [[String, String, String]] $\xrightarrow{get}$ [String] and [[String, String, String]] $\xrightarrow{find}$ [[String]]. The two model have been used to search one or a set of results in this graph. Therefore, queries are then expressed by the way of graph patterns as illustrated in the Figure 3.6 what also corresponds to the SPARQL statement:

$SELECT\ ?o\ WHERE\ ?o\ ri\ vi.\ ?o\ rj\ vj.$

The query of a specific edge such as $?o\ ri\ vi$ can be obtained with:

```
query D ri vi = [o| (o,r,v) <-d, r == ri,  v == vi]
```

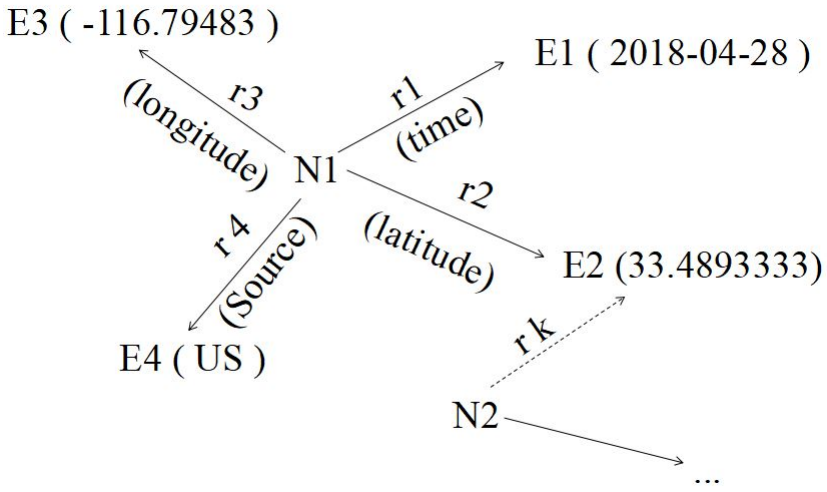


Figure 3.5: Sample graph.

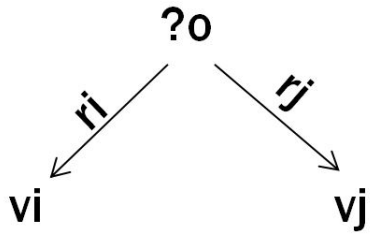


Figure 3.6: Sample query.

Functor and natural transformation

Having modeled n-ary relations by the way of a functor $[[x]]$, and having defined natural transformations establishing the iso-morphism with the CSV standard $-CSV \cong_{fromCSV} [[x]]$, we can now use the same model to represent graphs ; this, by adding an extra condition: $\forall db \in [[x]], \forall e \forall db, size(e) = 3$.. Thus, each element corresponds to a labeled edge (source,label,destination) or (subject,property,value) with a logical point of view of the graph/database. And if, list comprehension can be used for queries, it is more interesting to define a (human readable) query language able to find more complex information (e.g. join between other informations), and that can be used without knowing

anything about Haskell (i.e. queries are passed as a parameter of the compiled program). A directed graph can be represented by a set of edges and a functor $G(N) = \mathcal{P}(N \times N \times N)$ where N represents a set of nodes and $N \times N \times N$ edges of the form (source,label,destination). With a morphism $f : N \to N'$, we can define a graph morphism $m(f) = \mathcal{P}(f \times f \times f)$ that changes the nodes by preserving the structure of the graph - i.e. if (x, y, z) is an edge of g then $(f(x), f(y), f(z))$ is an edge of $m(g)$. $m(id_N)$ is a identity morphism, morphisms are composable (i.e. $m(f \circ g) = m(f) \circ m(g)$) what makes the set of graphs and morphisms another example of a category called $\mathcal{G}raph$.

Graphs play an important role in the BigData community and have many applications (see, for instance, Neo4J. Querying specific information then consists in finding a morphism from the graph representing a query to a graph database. Indeed, a query such as "(X has-Longitude -116.7948333) and (X has-latitude Y)" can be viewed as a labeled graph with two edges $\{Source : X \to US, latitude : X \to Y\}$, where X/Y denote variables as illustrated in Figure 3.7.

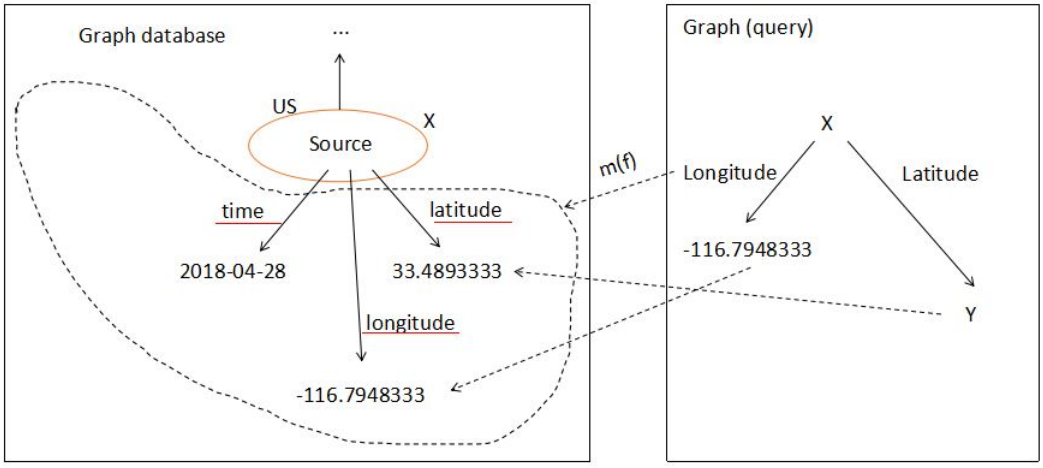


Figure 3.7: Sample graphs and query/morphism.

The query is a graph and an element of $G(N \cup X)$. The result is then a set of sets of pairs $\{\{(X, US), (Y, 33.4893333)\}\}$ and the program finding the

possible morphisms can be formalized by a function $unify : G(N') \times G(N) \rightarrow \mathcal{P}(\mathcal{P}(X \times N))$ $N' = N \cup X$ is the union of constant nodes N plus variables X. As a remark, $\mathcal{P}(X \times N)$ is here a shortcut for a mapping function $f : X \rightarrow N$, and is generally called "environment".

3.2.5 Transformations to SQL

To allow comparison this dataset has to be translated into SQL statements, by using the following program. Then, the result has been used to feed a RDBMS. Finally, the performance for an equivalent expression of q has been obtained with: $time\ (echo\ "SELECT\ *\ FROM\ db\ WHERE\ Source\ =\ "us"\ ;"\ |\ sqlite3\ database.db)$.

```
toSql db = concat [ "CREATE TABLE db
(src text,lbl text, dst text);\n", db']
where db' = concat (map (\e->concat
["INSERT INTO Db VALUES (",format e,");\n"]) db)

format []     = ""
format [x]    = concat ["'",x,"'"]
format (x:xs) = concat ["'",x,"',",format xs]

mkSql = do
dta <- fromCSV "usgs.csv"
writeFile "usgs.sql" (toSql dta)
```

3.2.6 Transformations to document database

The same approach has been considered for document database with the following translation program. More precisely, the dataset has been translated using the toMongo function below in Javascript statements stored in "usgs.js". This file is loaded by using the "load('usgs.js')" using the MongoDB console. Then the equivalent query for q is obtained with:

```
mongo --eval "db.store.find(lbl:'Resource', dst:'US').shellPrint()",
```

```
toMongo db = db'
    where db' =
       concat (map (\e->concat ["db.store.insert
       (",format e,")\n"]) db)
format [x,y,z] =
     concat ["{ src:'",x,"', lbl:'",y, "',dst:'",z,"' }"]
```

3.2.7 Transformations to graph

This dataset has been tested with the Neo4j graph database and its query language Cypher. First, the dataset has been loaded from the CSV file with the following command from the Neo4j interface.

```
LOAD CSV WITH HEADERS FROM "file:///usgs.csv"
AS line
MERGE (n:Node {name: line.src})
CREATE (m:Node {name: line.dst})
CREATE (n)-[:Link {name: line.lbl}]->(m)}
```

Next, the query is obtained with:

```
"MATCH ((x)-[r:Link \{Source:'US'\}]->(y)) WHERE y.Source='US'
RETURN *;" | cypher-shell -u user -p password}.
```

3.2.8 Transformation to key-Value and Column Oriented database

The result of the transformation can be viewed as a column model as illustrated in the Figure 3.8.

It corresponds to another functors $F3 = [(String, [(String, [String,])])]$.

The semantic function has to be adapted to this new data organization with a specific simple query as follows:

```
q' DB ri vi = [os | (r,vos) <- d,
                  r == ri,
                  (v,os) <- vos,
                  v == vi
                 ]
```

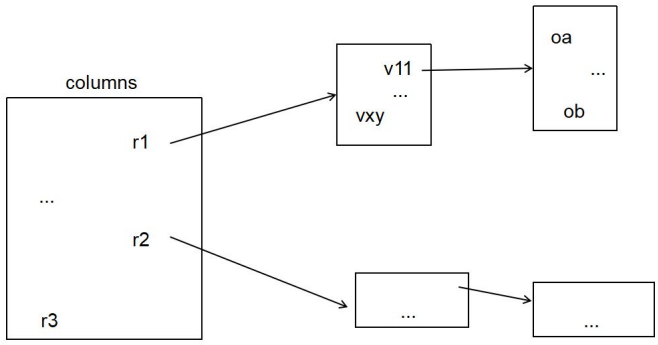


Figure 3.8: Sample of MonetDB model.

The semantic function for Cassandra is similar with MonetDB. Due to the nature of these two data structures, we use the same query and questioning methods as SQL.

- CSV file
- CSV file loading
- CSV file loading comparison
- Time to ”get” one item of information
- Time to ”get” specific information
- Time to ”find” information
- Time to ”find” information by its volume (size)
- The best choice to query information

Chapter 4

Application

4.1 CSV file

As explained in the previous part, morphisms representing lists (collections) and products (records) can be composed to model data structures in various ways. In particular, a table used in a relational database can be abstracted by a two dimensional array, i.e. [[x]] in Haskell where x represents the type of the cells. For our research, we have used the earthquake information which is downloaded from the United States Geological Survey. This CSV file contains 11236 rows and 9 different columns. These data records are seismic information from 14/04/2018 to 14/05/2018 in the world. In this file, the first line of earthquake information is from ”hv” (Burkina Faso). The first piece of information about ”us” is in the middle position of the file, and the first piece of information of ”nc” (New Caledonia) is in the latter position. In addition, the seismic information listed in this document also has the characteristics of large differences in the number of earthquakes occurring at the same location. For example, ”se”(Sweden) has 6 rows of data, ”uu”(Oceania) has 161 rows of data.

Based on the above series of features of the CSV file, we are divided into 7 main steps to analyse and discuss the data, and we will discuss these items step by step in the next section. In the following sections, all of the times' units are in ”ms”(Milliseconds). In our experiment, the time to obtain one ”get” result is

very fast, so we decide to get 500 times results. Then, we calculate the time for getting only one item of information. However, ”find” result is long enough to be directly gained by our program.

4.2 CSV file loading

Process	Minimum	Median	Maximum	Average
$CSV \rightarrow Table$	1140	1156	1203	1175
$CSV \rightarrow Table \rightarrow Map$	2171	2109	2359	2158
$CSV \rightarrow Map$	937	953	1156	1010
$CSV \rightarrow Table \rightarrow Graph$	2187	2203	2313	2237
$CSV \rightarrow Graph$	1031	1046	1109	1062

Figure 4.1: Time to load

As shown in Figure 4.1, the time to convert the CSV file format directly into map mode is the shortest. On the contrary, it takes the longest time to convert a CSV file into a table structure. Of course, if we follow the process described in the section of proposition, converting the CSV format to a table structure firstly, and then convert it to a map or graph format, the loading time will be greatly increased. This time is at least 2 times longer than the time to be directly converted to the corresponding model, due to the need to scan two times the amount of data. The other question is the number of objects that are different in the different model. For the Table model and Map model, there are more than 11000 objects. However, there are 98000 objects in the Graph model and it is still the fastest.

4.3 CSV file loading comparison

In this part, we have added the size of CSV file by 2 times, 5 times, 10 times 20 times, 50 times bigger than the original file. The time to load the corresponding file are in Figure 4.2. In addition, whole of the data is the average value which is obtained after 5 tests.

Figure 4.3 describes the line chart of Figure 4.2. This figure clearly shows

Process	1x	2x	5x	10x	20x	50x
$CSV \rightarrow Table$	1175	2468	6101	13652	23598	109964
$CSV \rightarrow Table \rightarrow Map$	2171	4410	11075	24670	47270	229770
$CSV \rightarrow Map$	937	1941	4974	11018	23671	119805
$CSV \rightarrow Table \rightarrow Graph$	2187	4864	12534	27708	54974	183511
$CSV \rightarrow Graph$	1031	2676	6432	14055	31375	73546

Figure 4.2: Time to load by different size of data

that as the amount of data increases, the time required to load each item is gradually increasing. When you reach 50 times bigger than the original file, the direct conversion time to the Graph model is the shortest. It takes the longest time to convert to Table model and then convert to Map model. In the other hand, Map model becomes slower than the table with a greater amount of data.

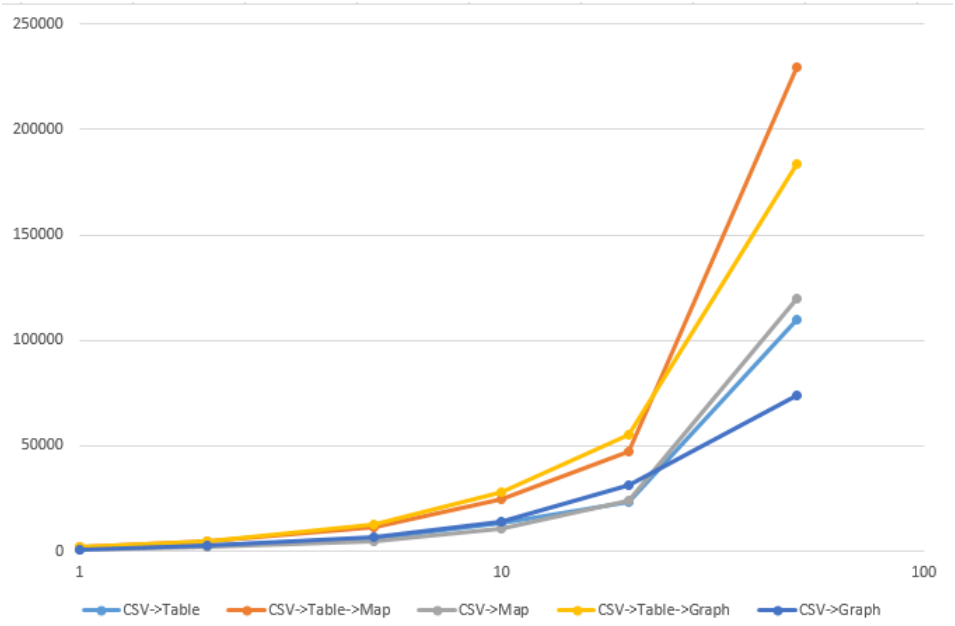


Figure 4.3: The comparison between the different volume of data by loading time

4.4 Time to ”get” one item of simple information

After finishing the work of inserting the CSV file, we use this data in memory to query some information. Here, we query one item of seismic information that occurs in ”us”, ”hv” and ”nc”. In the CSV file, ”hv” is located at the beginning of the file. ”us” is at the middle and ”nc” is in the end. The original CSV file is used here to retrieve information. Table 4.4 depicts the minimum, middle, maximum and average time to query the above information by table model, Map model and Graph model. The blue one is used by Table model, the red one is Map model and the green one is using Graph model. Figure 4.5 presents the Intuitive comparison chart for Figure 4.4.

	Table			Map			Graph		
Process	us	hv	nc	us	hv	nc	us	hv	nc
minimum	0.03525	0.021	0.077	0.028	0.018	0.053	0.0275	0.021	0.058
middle	0.04125	0.031	0.0625	0.03125	0.022	0.0625	0.03525	0.0275	0.0715
Maximum	0.0636	0.052	0.0812	0.0375	0.028	0.0812	0.045	0.035	0.087
Average	0.0467	0.035	0.074	0.03225	0.023	0.0632	0.036	0.028	0.0721

Figure 4.4: The comparison between three models by loading time

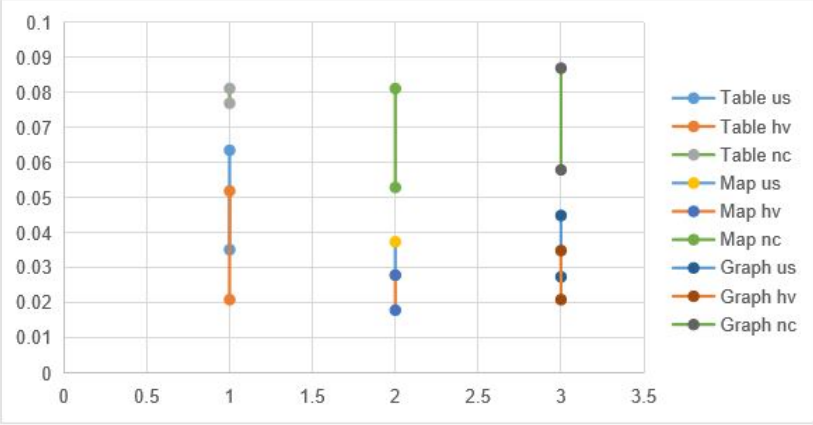


Figure 4.5: The comparison chart by loading time

Using the Table model to query the first seismic entry has used the longest

time. Conversely, the Map model is the shortest time to query a seismic information. It is about maximum 34% shorter than the Table model in terms of time usage. In addition, using the Graph model to query a seismic information saves about maximum 23% of the time compared to using the Table model. Therefore, Map model and Graph model can be used to optimize the time complexity of querying large databases. From Figure 4.5, we can see that the querying time for different information is different. One answer about the time difference to obtain the result could be in the position in the file of the first queried row.

4.5 Time to ”get” specific ordered information

We use the experiments in this section to determine whether the querying time is related to the location of the information to be queried in the file. To complete this experiment, we first reorder the information in the original file according to name of the ”Source” column. In this way, the information containing ”ak” is the first item of the new file, the information containing ”hv” is in the middle and the information containing ”uw” is at the end. Then, we flip the database again. The first item in the file is ”uw”, and the ”ak” information is at the end of the file. Figure 4.6 to 4.8 describes the minimum, middle, maximum and average time to query one ”ak”, ”hv” and ”uw” information(Ascending and Descending ordered information) by Table model, Map model and Graph model.

	Table				
	Ascending			Descending	
Process	ak	hv	uw	uw	ak
minimum	0.0625	0.0825	0.091	0.0616	0.102
middle	0.07225	0.09325	0.108	0.0742	0.12
maximum	0.09375	0.102	0.11975	0.092	0.133
average	0.0761	0.0925	0.106	0.076	0.118

Figure 4.6: Time to query one information by Table model

As all of the tables and figures shown, the time to search ”ak” is the most fastest one. Because it is the first information in CSV file. Then, as the location

	Map				
	Ascending			Descending	
Process	ak	hv	uw	uw	ak
minimum	0.0287	0.0467	0.0634	0.029	0.061
middle	0.03125	0.051	0.0672	0.03	0.0652
maximum	0.0345	0.057	0.0751	0.034	0.077
average	0.031	0.052	0.069	0.019	0.0677

Figure 4.7: Time to query one information by Map model

	Graph				
	Ascending			Descending	
Process	ak	hv	uw	uw	ak
minimum	0.025	0.049	0.0625	0.025	0.065
middle	0.03025	0.051	0.063	0.03	0.0691
maximum	0.0372	0.057	0.071	0.036	0.081
average	0.0309	0.052	0.066	0.0303	0.0717

Figure 4.8: Time to query one information by Graph model

of the information to be searched becomes more and more backward, the time takes longer and longer. The time to search ”uw” is the most slowest one. However, if we inverse the data set, ”uw” is the fastest one and the ”ak” is the slowest one. In summary, the query time is related to the location where the information to be queried is found. The searching speed for the information, whatever the model used, which is placed in the front position of the file is much faster than the others.

4.6 Time to ”find” information

In the previous section, we tested the time to insert a CSV file and query a seismic message. In this section, we test the time taken to query all earthquakes at the same location. We still test all seismic information that occurs in ”us”, ”hv” and ”nc”. We have used the original CSV file to complete these experiments. Figure 4.9 shows the minimum, middle, maximum and average time to query one ”ak”, ”hv” and ”nc” information by representative Table model, Map model and Graph model.

As shown in Figure 4.9, using the Map model is maximum 23% more ef-

	Table			Map			Graph		
Process	us	hv	nc	us	hv	nc	us	hv	nc
minimum	0.62	1.245	1.21	0.44	1.09	1.09	0.48	1.25	1.28
middle	0.88	1.88	1.36	0.69	1.41	1.25	0.74	1.56	1.33
Maximum	1.06	2.13	1.88	0.83	1.56	1.58	0.78	1.58	1.41
Average	0.85	1.75	1.48	0.66	1.35	1.31	0.67	1.46	1.34

Figure 4.9: Time to ”find” information by three models

ficient than using the Table model, and using the Graph model is maximum 17% more efficient than using the Table model. However, there is a factor which slows down the speed of query. ”hv” is before ”us” but still slower. In the CSV file, there are 2334 items of information about ”hv”, 1798 items about ”nc” and 1062 items about ”us”.

4.7 Time to ”find” information by its different volume

As you can see from Figure 4.10, using the same mathematical modeling method, the spending time to query different seismic information is different. Therefore, in this part, we re-adjust the original database. We reorder the file to put together seismic information at the same location at the beginning of the file. We found that the lowest count was for ”se” where only six earthquakes occurs and the largest was in ”uu” with 161 earthquakes. So, we adjust the seismic information of these two kinds of information to the first place of the CSV file. Then, test it to obtain the time to realize the ”find” information, which is similar in section 4.6. Figure 4.10 displays the time to query information by Table model, Map model and Graph model.

	table		map		graph	
	se	uu	se	uu	se	uu
minimum	1.72	1.89	1.47	1.69	1.56	1.87
median	1.83	1.96	1.56	1.78	1.68	1.95
maximum	2.5	2.99	1.89	1.94	1.90	2.14
average	2.02	2.28	1.64	1.80	1.72	1.99

Figure 4.10: Comparison between the two informations by three models

We discover that the time to search the information of ”uu” is at least 10% slower than ”se” by Map model and 16% slower than ”se” by Graph model. In summary, the time of querying seismic information at the same location is related to the amount of seismic information.

4.8 The best choice to query information

Our experiment is mainly divided into three parts, the first one is to load the csv file, and then to query seismic information. Finally, querying all of the seismic information that occurred at the same location. Because of this, the query time we calculated is: file loading time + time for querying. Figure 4.11 below is an example to explain the time to load the CSV file is different. If we directly load the CSV file by Map model and Graph model (which is shown in red line in Figure 4.11), the time is much shorter than the others.

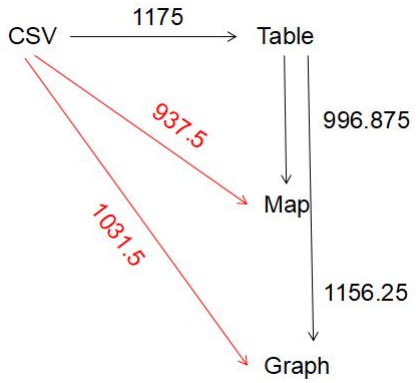


Figure 4.11: Comparison between the two information by three models

4.9 Real world application

In this section, we will compare the time to find all seismic information about ”us” using the five selected conventional databases and our approach. Figure 4.12 gives all the times. Surprisingly, our approaches are the fastest in all cases.

Mainly we measure that we are: 4+ times faster than RDBMS (Sqlite), 4+ faster than Key Store database (Cassandra), 10+ times than Column oriented database (MonetDB), 20+ times faster than a Document database (MongoDB) a huge time faster than a Graph database (NEO4J). Last, in our approaches, the Haskell functions based on Map and Graph models are the fastest of all.

Process	Time (ms)
RDBMS (Sqlite)	4
Column oriented database (MonetDB)	22.344
Key Store database (Cassandra)	4.6977
Document database (MongoDB)	10
Graph database (NEO4J)	307
Haskell by Table	0.85
Haskell by Map	0.66
Haskell by Graph	0.67

Figure 4.12: Comparison between different models

Conclusion

- Propositions' summary
- Perspectives

Chapter 5

Conclusion

5.1 Propositions' summary

This manuscript summarizes three years of study in the fields of Category Theory and functional programming applied to databases. The context of my work was introduced in chapter 1, mainly the fact that queries over databases can be carefully optimized in time if dedicated structures and requests can be precomputed. This led to the need of an analysis of the actual state in the database world. If many RDBMS exist since the 1970s, new kinds of databases arrive on the market. They all have specific behaviors and are specialized in certain kinds of processes. The second chapter takes time to present them and compares them together. Our goal is to propose a structured view of the databases and the queries based on mathematics. Therefore, we chose to root our researches on Category Theory, which is presented in the third chapter. In the Category Theory chapter, the two main concepts (Functors and Natural Transformations) are explained and the Monad is also presented for the best way to handle side effects. Haskell is chosen to implement all kind of collections of data, morphisms and functors. Its mathematical method of writing equations is fully adapted to our studies; we don't have to twist the equations as we have to do in case of Object Oriented Programming. We apply Category Theory to several ways of representing, at a high level, collections of data: such as table, list, map and graph. Several Functors are proposed to switch to one representation

to the other as well as the associated queries. Chapter three is our proposition of study : first we reduce the table behavior in a database to only three main representations: table, map and graph. Table structure can be used in case of non-indexed data, map structure can be used in case of indexed data and graph structure a special case recommended in case of a network of relations. The data set studied is presented and how we import it too. Next, we show some three representation chosen and the different Functors. Some special isomorphism are presented to target the native representation for dedicated databases. The next chapter, the fourth, is about the performances, mainly the time measurements. The time to load the initial data set using the defined procedure is commented upon and the degradation is measured in case of a growing of the initial dataset. The function get, which gets the first occurrence of information in the dataset is studied and compared according the three main defined structured. Some additional studies are carried out as position of the first entry seems important. The function find extracts a whole set of information and again we measure the time. As the amount seems important, again we take some measurements. Last, the dataset feeds the selected databases and again we measure the different times for a query. A comparison of the different obtained times shows that: functional programming in many cases is faster than convention database.

5.2 Perspectives

Our conclusion ends with a strange fact. A find expressed in a functional programming language seems better than a request on a conventional database. As for many years, databases are optimized in structures (btrees, etc.) and coded in imperative language such as C, etc. What could explain results and can this initial result be extended to other cases?

Here are some study propositions to go further one-step at a time.

The functional programming database proposed here loads all the data in memory and does all request in memory. Some database, such H2 or derby,

mimics this behavior, what are the performance results of these special kinds of databases (near to us, or the same as currently conventional)? What happens in case of distributed databases, how is the data distributed, etc. This could be the first step based on our first study.

The next way to explain this result can be in chosen query. The find just makes a lookup of the data, no updates and no inserts. Conventional databases have to deal with these insert and update queries so maybe they execute additional behaviors to protect data against update and insert which impact the time of a simple query. In case of a sudden stop, a database should store information to recover the state, what is the consequence on the time query of such a behavior. Enlarging the actual study, to insert and update queries, could show up some new interesting results.

After studying simple queries, why not consider complex queries such as a transaction. In transaction, several queries are bound together and either all can be done at one time or all have to be canceled. Transactions are harder to study than Monad for the state preservation needs to be extensively used.

Bibliography

[1] James Gro and Paul Weinberg. SQL The Complete Reference, 3rd Edition. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2010

[2] Peter Gulutzan and Trudy Pelzer. SQL Performance Turning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002

[3] Cansu Birgen, Heinz Preisig, John Morud, Advanced process simulation SQL vs NoSQL, Departement of chemical engineering norwegian university of science and technology, 2014

[4] Vatika Sharma, Meenu Dave, SQL and NoSQL Databases, International Journal of Advanced Research in Computer Science and Software Engineering, Department of CSE, Jagan Nath University, Jaipur, India, 2012

[5] B. DeCandia et al, "Dynamo: Amazon's Highly Available Key-Value Store", Proceedings 21st ACM SIGOPS Symposium on Operating Systems Principles, 2007

[6] Pankaj Sareen et al, NoSQL database and its comparison with SQL database, Computer Science Department, SPN College,Mukerian,International Journal of Computer Science and Communication Networks,Vol 5(5),293-298

[7] Jing Han, Haihong, E., Guan Le, Jian Du, "Survey on NoSQL database," Pervasive Computing and Applications (ICPCA), 2011

[8] Kristina Chodorow, MongoDB The Definitive Guide 2nd Edition, O'REILLY

[9] Zachary.Parker, S.V Vrbsky, Comparing NoSQL MongoDB to an SQL DB, The university of Alabama, USA, 2013

[10] Madalina. Croitoru, E. Compatangelo, On Conceptual Graph Projection, Technical Report AUCS/TR0403, 2004

[11] Amine.Ghrab, O.Romero, An Analytics-Aware Graph Database Modeling, Eura Nova RD, Mont-Saint-Guibert, Belgium, 2013

[12] C.Okasaki, Purely Functional Data Structures, Cambridge University Press, New York, USA, 1998

[13] Peter T.Wood, Query languages for graph databases, SIGMOD Rec,2012 Mont-Saint-Guibert, Belgium

[14] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner.Neo4J in Action. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2014.

[15] Olivier Cur and Guillaume Blin. RDF Database Systems: Triples Storage and SPARQL Query Processing. Morgan Kaufmann Publishers Inc., San Francisco, CA,USA, 1st edition, 2014.

[16] Maarten Vermeij, Wilko Quak, Martin Kersten, Niels Nes, MonetDB, a novel spatial column-store DBMS, TUDelft, OTB, section GIS-technology, The Netherlands

[17] Chunbin Lin, Data Compression in Database Query Processing, University of California, San Diego, La Jolla,CA,USA

[18] Weixiong Rao, MonetDB And The Application For IR Searches, UNIVERSITY OF HELSINKI. SEMINAR PAPER. COLUMN-ORIENTED SYSTEMS, 2012

[19] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures.IEEE Data Eng. Bull., 35(1):40–45, 2012

[20] P.A.Larson,E.N.Hanson, and S.L.Price.Columnar storage in sql server 2012.IEEE Data Eng. Bull., 35(1):15–20, 2012

[21] Mike Stonebraker et al., C-Store: A Column-oriented DBMS, MIT CSAIL,Cambridge, MA,USA

[22] Bob DUCHARm, Learning SPARQL, O'REILLY, 2011

[23] JavaScript tutorial, Tutorials points

[24] Markus Pilman, Kevin Bocksrocker, Fast Scans on Key-value, Snowflake Computing, Proceedings of the VLDB Endowment,Pages 1526-1537,2017

[25] Marc Seeger, Key-Value stores: a partical overview, Computer Science and Media Ultra-Large-Sites SS09, Stuttgart,Germany,2009

[26] Eben Hewitt, Cassandra The Definitive Guide, O'REILLY, 2011

[27] Arvid Arasu, Shivnath Babu, The CQL language Continuous Query Language: Semantic Fundations and Query Execution, Stanford University, 2003

[28] Lan Robinson, Graph Databases, O'REILLY, 2013

[29] J.A Bondy, Graph theory with applications, University of Waterloo, Canada, 1978

[30] Isnar Sumartono, Base64 Character Encoding and Decoding modeling, Faculty of Computer Science, Universitas Pembangunan Panca Budi, Jl. Jend. Gatot Subroto Km. 4,5 Sei Sikambing, 20122, Medan, Sumatera Utara, Indonesia,2016

[31] Brahim Medjahed, Generalization of ACID Properties, Departement of Computer and Information Science, The University of Michigan-Dearborn,2009

[32] Eric Brewer, CAP twelve years later: How the ”rules” have changed, University of California, Berkeley, 2012

[33] Laurent Thiry, Frédéric Fondement, Categorical reasoning about metamodels, Université de Haute Alsace, France, 2012

[34] Richard Bird, Oege de Moor, The Algebra of Programming, Prentice Hall, 1997

[35] Sreve Awodey, Category Theory, Oxford University Press,2006

[36] M.Barr and C.Wells,Category Theory for Computing Science, 2nd Ed. Prentice Hall International Ltd.,Hertfordshire, UK,UK,1995

[37] C.Okasaki, Purely Functional Data Structures, Cambridge University Press,New York,NY,USA,1998

[38] Joseph A.Goguen, A categorical manifesto, In Mathematical Structures in Computer Science, 1991

[39] Philip Wadler, Monads for functional programming, University of Glasgow, 2005

[40] Mac Lane, S. (1971). Categories for the Working Mathematician, volume 5 of Graduate Texts in Mathematics. Springer-Verlag. (xv, 126, 347, 357, 358, 415, 417)

[41] Goguen, Joseph A., Malcolm, Software Engineering with OBJ Algebraic Specification in Action, January 2000

[42] Jeremy Gibbons, Calculating Functional Programs, Springer−Verlag, 2002

[43] Laurent Thiry, Mariem Mahfoudh, and Michel Hassenforder. A functional inference system for the web. International Journal of Web Applications , 6(1):1–13, 2014

[44] Marlow S. A monad for deterministic parallelism. In Proceedings of the 4th ACM symposium on Haskell, Haskell '11, pages 71–82, New York, NY, USA, 2011. ACM

[45] Mac Lane S. Categories for the working mathematician. Graduate Texts in Mathematics. Springer, New York, second edition, 1978.

[46] Sugam Sharma, Udoyara S Tim,Johnny Wong, A brief review on leading big data models, Data Science Journal, Volume 13, 4 December 2014

[47] Peter Gacs, Boston University, Complexity of Algorithms, Lecture Notes, Spring 1999

[48] Granville Barnett, Luca Del Tongo, Data Structures and Algorithms: Annotated Reference with Examples, DotNetSlackers

[49] Martin Richards, Data Structures and Algorithms CST IB, CSTv II(General) and Diploma, Computer Laboratory University of Cambridge,2001

[50] Clifford A. Shaffer, A Practical Introduction to Data Structures and Algorithm Analysis Third Edition (Java) Department of Computer Science Virginia Tech Blacksburg, VA 24061, April 16, 2009

[51] Reema Thareja, Data Structures Using c, Department of Computer Science Shyama Prasad Mukherjee College for Women University of Delhi, Oxford university press, 2014

[52] Michael T. Goodrich, Roberto Tamassia, Data Structures and Algorithms in Java, WILEY, 2013

[53] Yuri Demchenko, Cees de Laat, System and Network Engineering Group University of Amsterdam, Defining Architecture Components of the Big Data Ecosystem, International Conference on Collaboration Technologies and Systems, 2014

[54] C. Okasaki. Purely Functional Data Structures. Cambridge University Press, NewYork, NY, USA, 1998.

[55] ORI SHALEV, Split-Ordered Lists: Lock-Free Extensible Hash Tables, Tel-Aviv University, Tel-Aviv, Israel [U+FF0C] Journal of the ACM (JACM) Volume 53 Issue 3, May 2006

[56] Hadley Wickham, The Split-Apply-Combine Strategy for Data Analysis, Rice University, Journal of Statistical Software, Volume 40, Issue 1. April 2011

[57] Nishant Shukla. Haskell data analysis cookbook. Packt Publ., Birmingham, 2014.

[58] Raghavendra Kune, Pramod Kumar Konugurthi, Arun Agarwal, Raghavendra Rao Chillarige, and Rajkumar Buyya. The anatomy of big data computing. Software, Practice and Experience, 46(1):79–105, January 2016.

[59] Donald Miner, Adam Shook, MapReduce Design Patterns, O'REILLY, 2013

[60] Maarten M. Fokkinga. Calculate categorically! Formal Aspects of Computing, 4(1):673– 692, Nov 1992.

[61] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA '95, pages 324–333, New York, NY, USA, 1995. ACM.

[62] Laurent Thiry and Michel Hassenforder. A calculus for (meta)models and transformations. International Journal of Software Engineering and Knowledge Engineering, 24(5):715–730, 2014.

[63] David E. Rydeheard and Rod M. Burstall. Computational Category Theory. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1988.

[64] Tatsuya Hagino. A Categorical Programming Language. PhD thesis, 1987.

[65] David I. Spivak. Ologs: a categorical framework for knowledge representation. CoRR, abs/1102.1889, 2011.

[66] Ryan Wisnesky. Functional Query Languages with Categorical Types. PhD thesis, Cambridge, MA, USA, 2014.

[67] Simon Peyton Jones and Philip Wadler. Comprehensive comprehensions. In Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop, Haskell '07, pages 61–72, New York, NY, USA, 2007. ACM.

[68] Hal R. Varian. Big Data: New Tricks for Econometrics, Journal of Economic Perspectives—Volume 28, Number 2 Spring 2014 Pages 3 28.