A novel algorithm for transforming row-oriented databases into column-oriented databases

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science and Engineering

Submitted By
Rupinder Kaur
(Roll No. 801132024)

Under the supervision of:
Ms. Karamjit Kaur
Lecturer

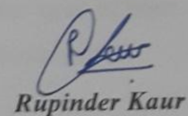
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2013

Certificate

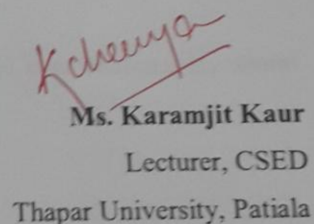
I hereby certify that the matter which is being presented in the thesis entitled, **"A novel algorithm for transforming row-oriented databases into column-oriented databases"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Ms. Karamjit Kaur* and refers others researcher's works which are duly listed in the reference section.

The matter represented in this thesis has not been submitted for the award of any other degree of this or any other university.



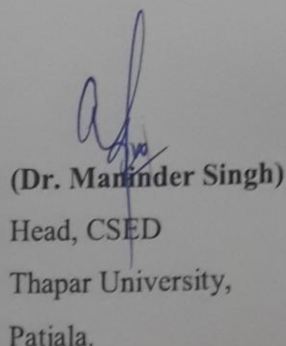
Rupinder Kaur

This is to certify that the above statement made by the candidate is correct and true to best of my knowledge.

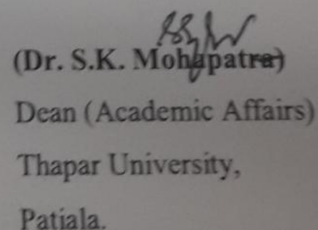


Ms. Karamjit Kaur
Lecturer, CSED
Thapar University, Patiala

Countersigned by



(Dr. Maninder Singh)
Head, CSED
Thapar University,
Patiala.



(Dr. S.K. Mohapatra)
Dean (Academic Affairs)
Thapar University,
Patiala.

Acknowledgement

I express my sincere and deep gratitude to my guide *Ms. Karamjit Kaur*, Lecturer in Computer Science & Engineering Department, for the invaluable guidance, support and encouragement. She provided me all resource and guidance throughout thesis work.

I am thankful to *Dr. Maninder Singh*, Head of Computer Science & Engineering department Thapar University Patiala, for providing us adequate environment, facility for carrying thesis work.

I would like to thank to all staff members who were always there at the need of hour and provided with all the help and facilities, which I required for the thesis work.

I would also like to express my appreciation to my friends and classmates for helping me in the hour of need and providing me all the help and support for completion of my thesis.

I am deeply indebted to my family for the inspiration and ever encouraging moral support, which enabled me to pursue my studies.

Rupinder Kaur
801132024

Abstract

With the development of cloud computing and distributed system, more and more applications are started to migrate to the cloud to use its computing power and scalability. These systems require interaction of heterogeneous data and applications that involves databases that store the data of the common domain under different representations. The interactions are result of schema mappings and transformation that is also called data migration. The above interaction of different systems and integration is known as information integration.

Two well-known approaches to information integration are data exchange and data integration. In the first approach, data is extracted from multiple heterogeneous sources, restructured into a common format, and finally transformed into a target schema. In second approach, several local databases are queried by users through a single, integrated global schema. Mappings are determined to know, how the queries that users pose on the global schema are to be reconstructed in terms of the local schemas.

In this thesis, a novel approach is proposed that transforms a relational database into a column-oriented database. Out of various available column-oriented databases like HBase, Cassandra etc., we have considered HBase for the study, which is an open-source distributed database and similar to BigTable which is the database proposed and used by Google. For relational database, we have considered PostgreSQL. The transformation procedure is divided into two phases. First relational schema is transformed into HBase schema based on the data model of HBase. Using four rules, HBase schema is extracted from PostgreSQL schema which could be further utilized to develop an HBase application. In the second phase, relationships between elements of two schemas are expressed as a set of nested schema mappings, which could be further used to create a set of nested queries that transform the PostgreSQL database into the HBase database representation automatically.

Table of Contents

Certificate..i

Acknowledgement..ii

Abstract...iii

Table of Contents..iv

List of Figures...vi

List of Tables..vii

Chapter 1 Introduction...1

1.1 Storage Solution……………………………………………….2

1.2 Relational Database……..……………………….……………3

1.3 NoSQL Databases……………………………….……………4

1.4 Relational Vs NoSQL Model…………………………………7

1.4.1 Shortcomings of relational databases………………...8

1.4.2 NoSQL Benefits Relational databases……………….9

1.5 Classes of NoSQL Databases………………………………..10

1.5.1 Key-value Databases…………………...…………10

1.5.2 Column-family Databases……………………………11

1.5.3 Graph Databases...12

1.5.4 Document-Oriented Databases......................................13

1.6 Objective...14

1.7 Thesis Outline...15

Chapter 2 Column-Family Databases..16

2.1 Row-oriented systems………………………………………..16

2.2 Column-oriented systems……………………………………18

2.3 Benefits of column-oriented databases………………………19

2.4 Available Column-oriented Databases………………………19

2.4.1 HBase…………………………………………………...20

2.4.2 Cassandra……………………………………………….20

2.4.3 Comparison between Cassandra and HBase…………21

2.5 HBase...22

2.5.1 History..23

2.5.2 Data Model....23
2.5.3 Data Versioning....25
2.5.4 Data Modelling....25
2.5.5 Nested Entities…....26
2.5.6 Business Use-cases....27
Chapter 3 Literature Survey on schema mapping....30
3.1 Schema Mapping Models 31
3.1.1 Match-Driven Model....31
3.2.2 Example-Driven Model....32
3.2 Popular Schema Mapping tools....32
3.2.1 Clio....32
3.2.2 Clip....33
3.2.3 Eirene....33
3.2.4 Mweaver....35
3.3 Representing Schema Mappings....36
3.3.1 GLAV....36
3.3.2 Tableaux....37
3.3.2.1 History....38
3.4.2.2 Relational Tableaux....39
3.4.2.3 Tuple Generating dependencies....39
Chapter 4 Problem Statement and Motivation....41
Chapter 5 Design and Implementation....42
5.1 Example Database Overview....43
5.1.1 Schema of example database....44
5.2 Transformation and Mapping....44
5.2.1 Transformation of Schema....44
5.2.2 Schema Mapping....40
5.2.2.1 Element Correspondence....52
5.2.2.2 Tableaux....55
5.3.2.3 Elements Mapping....55
5.3 Target HBase Database in the form of Tables....61
Chapter 6 Conclusion and Future Scope....63
References....65

List of Figures

Figure 1.1	CAP Theorem....................5
Figure 2.1	Column-oriented vs. row-oriented storage layout....................18
Figure 2.2	HBase Data organisations....................20
Figure 2.3	Cassandra Data organisation....................21
Figure 2.1	The coordinates used to identify data in an HBase table are B rowkey, C column family, D column qualifier, and E version....................24
Figure 2.2	Nesting entities in an HBase table....................27
Figure 3.1	Visual specifications of value correspondences....................33
Figure 3.2	Workflow for interactive design and refinement of schema mappings via data examples....................34
Figure 3.3	Eirene: System Architecture 34
Figure 3.4	A comparison between the Match-driven approach and the Sample-driven approach 35
Figure 3.5	A screenshot of MWEAVER. Left: The input spreadsheet. Right: The expanded list of candidate mappings....................44
Figure 5.1	HBase data model....................45
Figure 5.2	Schema of source database....................45
Figure 5.3	Transformation overview....................46
Figure 5.4	Flowchart of schema transformation....................47
Figure 5.5	Target schema transformation after rule 2....................49
Figure 5.6	Target Schema transformation after rule 3....................51
Figure 5.8	Schema correspondences between source schema and target schema....................54
Figure 5.8	Tableaux hierarchy of source and target....................58

List of Tables

Table 2.1 Comparison between HBase and Cassandra..22
Table 5.1 Target table and respective rowkeys..48
Table 5.2 Post table...61
Table 5.3 Tag table..62
Table 5.4 Comment table..62
Table 5.5 User table...62

Chapter 1: Introduction

The "online" systems of the 1980s have gone through a series of intermediate transformations. Today's web and mobile applications are result of these transformations. Now, these systems are capable of solving new problems for larger user population and the computing infrastructure on which these systems run, have changed drastically.

The architecture of these software systems has likewise transformed. A modern Web application can support millions of concurrent users by spreading load across a collection of application servers behind a load balancer. Changes in application behavior can be rolled out incrementally without requiring application downtime by gradually replacing the software on individual servers. Adjustments to application capacity are easily made by changing the number of application servers.

Modern web applications may encounter millions of concurrent users and balance load across a cluster of application servers which are placed behind a load balancer. Google and Amazon both employ distributed databases for use by web services that implement data stores. Their data format and layout can be dynamically controlled by the application.

But database technology has not kept pace. Relational database management systems (RDBMS) developed in the 1970s, have traditionally been employed by the enterprise business applications and commercial sector for mission critical. These applications commonly require high-throughput transaction processing and fault tolerance [1]. Although highly appropriate for the enterprise application domain, recent development in distributed database technologies and web technologies have forced inventors and developers to switch to new alternatives.

Google and Amazon were the first to invent new alternatives to data management in the lack of commercially available alternatives. These non-relational database technologies are fulfilling all the needs of modern software systems. Developing a new database is not feasible, as its maintenance and support is difficult to handle by small companies. Commercial suppliers of non-relational database technology have also emerged to provide data solution that enables the cost-effective management of data behind modern Web and mobile applications.

1.1 Storage Solution

The term database is used to describe a variety of systems employed to organize storage of data. There are different types of systems as well as models that do this in different ways. There has been a significant amount of work done on column-oriented database systems. Column database stores data tables as sections of columns of data rather than as rows of data. These database systems have been shown to perform more than an order of magnitude better than traditional row-oriented database systems on analytical workloads. Column-stores are more efficient for read-only queries since they only have to read from disk or from memory those attributes accessed by a query.

The development of database technology can be divided into three eras based on data model: navigational, relational, and post-relational. The two early navigational data models were the hierarchical model that was first developed as a part of NASA Apollo moon project in 1964 [2] and IBM joined NASA to develop the first hierarchical model named IMS in 1966 and Network model was implemented by General Electric under the guidance of scientist Charles Bachman who named it IDS.

The relational model was first proposed by E. F. Codd in 1970 [1]. System R became the basis for IBM's first commercially available relational database management system SQL/DS, which was announced in 1981. Another early commercial database management system Oracle was developed in the late 1970s using SQL [4] as its language. Thereafter, hundreds of relational database management systems have incorporated SQL [4].

Since the beginning of computer software, the data that the applications had to deal with has grown in complexity enormously. Complexity of the data included not only its size, but also the inter-connectedness of the data, its ever-changing structure and concurrent access to the data. A number of new storage technologies have been created, with one common goal to solve the problems in which relational databases were not good at. To solve several needs, a variety of new types of databases have appeared. In general, these new databases are very different with traditional relational databases, so it is referred to as "NoSQL" databases.

1.2 Relational Databases

For several decades, the most common data storage model has been the relational model. Dr. E.F. Codd [1] of IBM is known as the father of relational databases. IBM developed the first prototype of relational database and standardized it by ANSI in 1986. The first relational database was released by Relational Software and it is later named as Oracle.

A relation is defined as a set of tuples having the same attributes. One of these tuples represents a real world entity and its related information. A set of one or more attributes has to be chosen as the primary key, which must be distinct among all tuples in the relation [4]. The relation itself is usually described as a table where each tuple corresponds to a row and each attribute to a column. Both rows and columns are unordered. Relations can be modified using insert, delete and update operations. They can also be queried for data by operators to identify tuples, columns and combine relations. The most common way to query database is using the structured query language.

SQL is structured Query Language which is a query language for storing, manipulating and retrieving data stored in relational database. It is an ANSI standard but there are many different versions of the SQL language. All relational database management systems like MySQL [11], MS Access and Oracle [10] use SQL as standard database language. SQL also allows users to describe, define, access, and manipulate the data and provides easy method to create view and set permissions on tables, procedures, and views.

Relational databases are still widely used but in some cases the relational database technology has shown its age. One issue that has been widely debated is the object relational impedance mismatch, which explains the fact that the relational model is conceptually incompatible with the object oriented model that is highly dominant in software engineering today. One of the most powerful characteristics of relational databases, strict enforcement of referential integrity and data conformity, is putting restriction to changing business requirements.

Relational databases also suffer from runtime problems in many common scenarios and show very poor performance for sparse tables. Sparse tables are tables where many rows have no values for many columns. Sparse table is a type of semi-structured data where the data has many optional attributes. But the relational model

is built to handle structured data only i.e. data with a fixed and complete schema. Stefano Mazzocci of Apache and MIT research fame states the above problem as, “The great majority of the data out there is not structured and there is no way in the world you can force people to structure it" [1].

Many data sets are naturally ordered in networks like the semantic web movement that generates gigantic data set, same is the case with bio-informatics, social networks, artificial intelligence, business intelligence and many more. In relational data model the problem is the need of join operation on multiple tables but the join operations are very expensive. Every jump along an edge means one join. Generally most applications stop looking after three jumps as the exponential processing cost is very high. There are many limitations of relational databases, few are outlined below:

1. The static, rigid and inflexible nature of the relational model makes it difficult to evolve schemas.
2. The relational model is very poor in dealing with semi-structured data, which going to be the next big thing in information management.
3. The relational model can store network oriented data, but it is very weak in traversing the network in order to extract information.
4. Correlated Multi-tables which exists in relational database became the major setback for database scalability.
5. Existing relational database cannot support big data in search engine.

1.3 NoSQL Databases

The term NoSQL was first introduced in 1998 for a relational database that restricted the use of SQL. The term was emerged again in 2009 and used for conferences of advocates on non relational databases. These conferences state that especially Web 2.0 start-ups have begun their business without Oracle [10] and MySQL [11]. Instead, they built their own datastores influenced by Amazon’s Dynamo and Google’s Bigtable in order to store and process huge amounts of data. Most of these datastores are open source software. For example, Cassandra which is a column-oriented database originally developed for a new search feature by Facebook is now part of the Apache Software Project.

Different types of data stores satisfy different needs and hence various non-relational databases are classified into key-value, columnar, document-oriented and graph-based databases. Document oriented databases gain an immense ease-of-use, while the key-value and column oriented databases make it easier to distribute data over clusters of computers. Graph-bases databases are preferred in applications where relationships amongst data are as important as data.

Brewer's CAP theorem [7] explains various possible trade-offs. The theorem states that it is difficult for a web service to provide guarantee for consistency, availability and partition tolerance at the same time.

Consistency means all the nodes of a system should have the same data. *Availability* means that the data should be available all the time even if some nodes are failed.

Partition tolerance makes the system continues to operate despite arbitrary message loss.

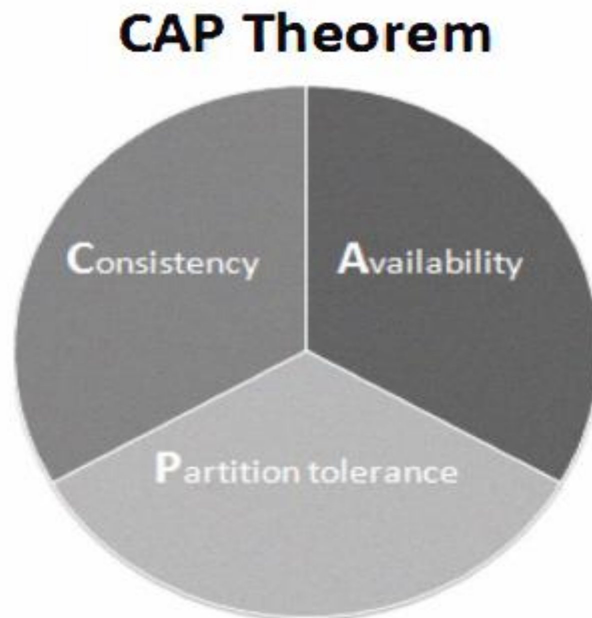


Figure 1.1: CAP Theorem [7]

In web applications based on horizontal scaling strategy, it is necessary to decide between C and A that is Consistency and Availability. Usually DBMSs prefer C over A and P. There are two directions in deciding whether C or A. One of them requires strong consistency as a core property and tries to maximize availability. The advantage of strong consistency, that is ACID transactions, means to develop

applications and to manage data services in more simple way. On the other hand, complex application logic has to be implemented, which detects and resolves inconsistency.

The second direction prioritizes availability and tries to maximize consistency. Priority of availability has rather economic justification. Unavailability of a service can imply financial losses. Existence of 2 phase commit protocol ensures consistency and atomicity from ACID [9]. Then, based on the CAP theorem, availability cannot be always guaranteed and for its increasing it is necessary to relax consistency, that is when the data is written, not everyone, who reads something from the database, will see correct data. This is called eventual consistency or weak consistency. Such transactional model uses BASE (Basically Available, Soft state, Eventually Consistent) model.

BASE is an acronym for Basically Available Soft-state services with Eventual-consistency. BASE is associated with No SQL data stores that focuses on Partition tolerance and availability and literally chucks consistency out in order to achieve better partitioning and availability.

Although relational databases are very mature, but upcoming NoSQL databases have few advantages over traditional relational databases which are listed below:

1. Relational databases follow strict data consistency. But the ACID properties implemented by RDBMSs might be more than necessary for particular applications and use cases. As an example, Adobe’s Connect Now holds three copies of user session data; these replicas do not neither has to undergo all consistency checks of a relational database management systems nor do they have to be persisted. Hence, it is fully sufficient to hold them in memory.
2. NoSQL databases provide significantly higher performance than traditional RDBMSs. For example, the column-store Hypertable which follows Google’s Bigtable approach allows the local search engine to store one billion data cells per day. Take another example, Google’ MapReduce approach is able to process 20 petabyte a day stored in Bigtable.
3. NoSQL databases are designed in a way that PC clusters can be easily and cheaply expanded without the complexity and cost of sharding which involves cutting up databases into multiple tables to run on large clusters or grids.

4. In contrast to relational database management systems most NoSQL databases are designed to scale well in the both directions and not rely on highly available hardware. Machines can be added and removed without causing the same operational overhead involved in RDBMS cluster-solutions.

1.4 Relational Vs NoSQL Model: An overview

For a quarter of a century, the relational database has been the dominant model for database management. But today, NoSQL databases are gaining mind-share as an alternative model for database management. Relational databases are great for enforcing data integrity. They are the tool of choice for online transaction processing (OLTP) applications like data entry systems or on-line ordering applications.

Following are few strong points of relational databases which made it popular since 1970s and are still widely in use:

1. ACID transactions are most definitely required in certain use cases. Databases used by banks and stock markets for example, always must give correct data. Where money is concerned, making predictions is not allowed. Though, it is true that it does not matter how much minutes does a tweet made, show up in twitter feed. But the same cannot be said for a billing system or accounting database.
2. SQL is a common language and switching from one database to another can be easily done. Secondly, a large number of tools and libraries are available t o interact with the data.
3. The R in RDBMS traces its history back to research by E. F. Codd published in the June 1970 issue of Communications of the ACM [1]. Since then, it has been expanded upon, improved and clarified. The relational model for databases is so popular because it is an excellent way to organize information. It maps very well to an enormous variety of real-world data storage needs, and when properly normalized, it is fast and efficient.

1.4.1 Shortcomings of Relational databases

Relational databases require data to be normalized so that it can provide quality results and prevent redundant data. It uses the concept of primary, secondary keys and indexes to enable queries to retrieve data quickly. But this comes with a cost; normalizing data requires more tables, which requires more table joins, thus requiring more keys and indexes. As databases start to grow into terabytes, performance starts to significantly fall off. Often, hardware is thrown at the problem which can be expensive both in terms of capital, ongoing maintenance and support.

Following are few disadvantages using relational model:

1. Relational databases allow versioning or activities like create, read, update and delete. For databases, updates should never be allowed because they destroy information. Rather when data changes, the database should just add another record and duly note the previous value for that record.
2. Performance falls off as relational databases normalize data. Since, normalization requires more tables, table joins, keys and indexes and thus more internal database operations for implementing queries.
3. Users can scale a relational database by running it on a more powerful and expensive computer. To scale beyond a certain point, it must be distributed across multiple servers. Relational databases do not work easily in a distributed manner because joining tables across a distributed system is difficult. Also, relational databases are not designed to function with data partitioning, so distributing their functionality is a difficult task.
4. With relational databases, users must convert all data into tables. When the data does not fit easily into a table, the databases structure can be complex, difficult and slow to work with.
5. Using SQL is convenient with structured data. However, using this language with other types of information is difficult because it is designed to work with structured, relationally organized databases with fixed table information. SQL can entail large amounts of complex code and does not work well with modern agile development.

1.4.2 NoSQL databases benefits over Relational databases

The NoSQL approach presents huge advantages over relational databases because it allows application to scale to new levels. The new data services are based on truly scalable structures and architectures, built for the cloud, built for distribution and are very attractive to the application developer. There is no need for Database Administrator (DBA), no need for complicated SQL queries and it is fast.

Following are the few advantages, NoSQL offers over RDBMS [28]:

1. *Elastic scaling:* For years, database administrators have relied on scale up, that is buying bigger servers and distributing the database across multiple hosts as load increases. However, as transaction rates and availability requirements increase, the economic advantages of scaling out on commodity hardware become irresistible. RDBMS might not scale out easily on commodity clusters, but the new breed of NoSQL databases are designed to expand transparently to take advantage of new nodes.
2. *Big data:* Just as transaction rates have grown out of recognition over the last decade, the volumes of data that are being stored also have increased massively. Relational database's capacity has been growing to match these increases, but as with transaction rates, the constraints of data volumes that can be practically managed by a single RDBMS are becoming intolerable for some enterprises. Today, the volumes of “big data" that can be handled by NoSQL systems such as Hadoop, cannot be handled by the biggest RDBMS.
3. *No DBAs:* High-end RDBMS systems can be maintained only with the assistance of expensive, highly trained DBAs. DBAs are intimately involved in the design, installation, and ongoing tuning of high-end RDBMS systems. NoSQL databases are generally designed to require less management, such as automatic repair, data distribution and simpler data models lead to lower administration and tuning requirements.
4. *Economics:* NoSQL databases typically use clusters of cheap commodity servers to manage the exploding data and transaction volumes, while RDBMS tends to rely on expensive proprietary servers and storage systems. The result is that the cost per gigabyte or transaction/second for NoSQL can be many

times less than the cost for RDBMS, thus allowing to store and process more data at a much lower cost.

5. *Flexible data models:* Even minor changes in the relational model have to be carefully managed and may necessitate downtime or reduced service levels. NoSQL databases have far more relaxed or even nonexistent data model restrictions. Application and database schema changes are not have to be managed as one complicated unit.

1.5 Classes of NoSQL Databases

NoSQL databases can be categorized into four classes: Key-value databases, which implement a key to value persistent map for data retrieval, Column-family databases, which is inspired by BigTable model of Google, Document oriented databases, which are oriented to store semi-structured data and Graph Databases, which are oriented to store data in graph-like structures.

1.5.1 Key-Value Databases

Key-value databases have very simple data model. Rather than tables or semi-structured documents, data is just organized as an associative array of entries. A unique key is used to identify a single entry and all of the three available operations i.e. delete the given key, change the entry associated with the key and insert a new key, uses this key. While the key-value data store looks and acts like an associative array, it may rely on tables, indexes and other artifacts of relational systems to be efficient in practice.

Key-value data stores use a data model similar to the popular memcached (distributed in-memory cache), with a single key-value index for all the data. Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering. Key-value data stores allow the application to store its data in a schema-less way [10]. The data could be stored in a data-type of a programming language or an object. Because of this, there is no need for a fixed data model. Besides the data-model and the API, modern key-value stores favour high scalability over consistency and therefore most of them also omit rich ad-hoc querying and analytics features (especially joins and aggregate operations are set aside). Often, the length of keys to be stored is limited to a certain number of bytes while there is less limitation on values ([9], [10]).

Key-value data stores have existed for a long time (e.g. Berkeley DB [6]) but a large number of this class of NoSQL stores that has emerged in the last couple of years has been heavily influenced by Amazon`s Dynamo. Key-value data stores are best suited for following type of application [7]. Suppose, there is need to develop a discussion forum software which have a home profile page that gives the user's statistics (no. of messages posted, etc) and the last ten messages posted by them. In this case the page is read from a key that is based on the user's id and retrieves a string of JSON that represents all the relevant information, where JSON (JavaScript Object Notation) is a lightweight data interchange format. A background process recalculates the information after every 15 minutes and writes to the data store independently [5]. Popular Key-value databases are Dynamo [19], SimpleDB [14], Redis [15].

Redis is a new project on Key-value memory database. Redis load entire data into memory for faster access and save the data asynchronously to the hard disk. Performance of Redis is better as compared to other databases due to its pure memory operation. Redis supports data structures like list and set and various related operations. The main limitation is limited size of the database by physical memory, so Redis cannot be used for massive storage, and its scalability is poor.

1.5.2 Column-oriented Database

Column-oriented data stores were created to store and process very large amounts of data distributed over many machines. There are still keys but they point to multiple columns. The columns are arranged by column family, where a column family tells how the data is stored on the disk. All the data in a single column family will reside in the same file. A column family can contain columns or super columns, where as super column is a dictionary; it is a column that contains other columns. These extensible record stores have been motivated by Google's success with BigTable, where BigTable is a NoSQL data store introduced by Google in 2006 [5]. Basic data model is rows and columns, and basic scalability model is splitting both rows and columns over multiple nodes. A column family is a container for rows, analogous to the table in a relational system. Each row in a column family can reference by its key. Columns and super columns in a column database are spare, meaning that they take exactly 0 bytes if they do not have a value in them.

Rows are split across nodes through sharding on the primary key, where sharding [9] is a technique which involves partitioning data by rows across a number

of distributed servers. They typically split by range rather than a hash function. It means queries on a range of values do not have to traverse every node. Columns of a table are distributed over multiple nodes by using column groups". Column groups are simply a way to indicate which columns are best stored together. Horizontal and vertical partitioning can be used simultaneously on the same table.

Column-oriented data stores are the preferred method for storing time series data in many applications in capital markets. For example, they are excellent choices for storing tick data from stock exchanges, where tick data refers to market data which shows the price and volume of every print, also often includes information about every change to the best bid and ask. The leading technologies in this area are Google's BigTable [53] and Cassandra [16], originally developed by Facebook.

HBase is a column oriented database inspired by Google's BigTable design. HBase is accessed through a java API or through a REST and thrift gateway. The data model of HBase can be seen as a four dimensional dictionary. The keys of this dictionary are, in order: Table, Row, Column, Time-stamp, which together points to a value.

1.5.3 Graph Databases

Leading the innovation in Web 3.0 are websites like Facebook, Twitter and LinkedIn. The amount of data generated by these websites in a month is several terabytes. To traverse this huge data various scientific communities have come together to give a common solution. Graph databases like Neo4j [2] and FlockDB [57] are the end results which provides traversal of millions of nodes in milliseconds which otherwise would have taken hours in traditional RDBMS. In graph based data stores, instead of tables of rows and columns and the rigid structure of SQL, a flexible graph model is used which can scale across multiple machines.

These kinds of databases are for data whose relations are well represented with a graph-style that is where elements are interconnected with an undetermined number of relations between them. In graph based data store, every element contains a direct pointer to its adjacent element and no index look-ups are necessary [10]. General graph databases that can store any graph are distinct from specialized graph databases such as triple stores and network databases. Most of the important information is stored in the edges. Meaningful patterns emerge when one examines the connections and interconnections of nodes, properties and edges. Graph based data store is the

backbone for all social websites. Common Graph databases are Neo4j [2], OrientDB [12], Titan [17].

OrientDB is an open source database with combined features of document and graph databases. Though it is a document-based database, the relationships are represented as in graph databases with direct connections between records. It supports ACID transactions. Query language is an extension of SQL that handles relationships without joins. It is written in java and can operate in schema-less, schema-full or a mix of both.

Neo4j is one of the leading open-source and high-performance graph databases supported by Neo Technology. It has been in production for more than 5 years. Neo4j is fully transactional [2], embedded and disk-based persistence engine that is capable of storing data in graph structure not in tables. Neo4j is exceptionally scalable and has easy to use API that provides smooth traversal. Trade-off of transactional attributes for scalability and better performance has been the common approach in NoSQL technologies. It provides ACID properties [13] support which makes it unique and strong.

1.5.4 Document-Oriented Databases

A document database stores, retrieves, and manages semi-structured data. The element of data is called document. Unlike the key-value stores, these systems generally support secondary indexes and multiple types of documents (objects) per database. Like other NoSQL systems, document data stores do not provide ACID transactional properties. Compared to relational databases, collections correspond to tables and documents to records. But there is one big difference, every record in a table have the same number of fields, while documents in a collection could have completely different fields. Documents are addressed in the database via a unique key that represents that document.

Document data stores are usually implemented without tables. This allows client application to add and remove attributes to any single tuple without wasting space by creating empty fields for all other tuples. All tuples can contain any number of fields, of any length. Even if used in a structured way, there are no constraints or schemas limiting the database to conform to a preset structure. Hence, the application programmer gains ease of use and the possibility to create very dynamic data. A document data store can be implemented as a layer over a relational or object

database. Another option is to implement it directly in a semi-structured file format, such as JSON [5]. Document data store implementations are a cross between a key-value pair and record structure paradigms. There are many enhancements to the structure that can be done like d*ynamic restructuring of the schema and representation of non-structured data.

One of the scenario in which document data stores can be preferred is explained here. Suppose, there is a requirement of database which creates profiles of refugee children with the aim of reuniting them with their families. The details that need to be recorded for each child vary tremendously with circumstances of the event and they are built up gradually, for example a young child may know their first name and a picture of them can be taken but they may not know their parent's first names. Later, a local may claim to recognize the child and provide additional information that can be recorded but until the information is verified, it is treated sceptically [5]. Common document databases are MongoDB [18], CouchDB [18].

MongoDB is an open source, high-performance, scalable document-oriented database. Its development was started in October 2007, and is still an active project. MongoDB stores data in binary JSON-style documents. It does not support strong transactional requirements but allows scaling out on different servers.

1.6 Objective

Following are the objectives of this thesis:

1. To study and analyse various available NoSQL databases and understand the classification of NoSQL databases.
2. To gain in-depth knowledge of Column-oriented databases and compare column-oriented databases with row-oriented databases.
3. Choose a column-oriented database by comparing available column-oriented databases and study the chosen column-oriented databases.
4. To study various available schema mapping mechanisms.
5. To propose an algorithm for transforming a relational database to a column-oriented database.
6. To explain the working of the proposed algorithm using a use case.

1.7 Thesis Outline

The rest of the thesis is organised as follows:

Chapter 2 covers the introduction and comparison of column-oriented databases with row-oriented databases. It discusses about two popular column-oriented databases namely, HBase and Cassandra and a brief comparison of both is provided.

Chapter 3 covers background information on various schema mapping mechanisms and transformation and various popular mapping tools.

Chapter 4 covers the motivation and the problem statement for the thesis work.

Chapter 5 describes the design and implementation of the proposed algorithm for schema mapping and transformation of a relational database (PostgreSQL) to a column-oriented database (HBase). Working of the proposed algorithm is explained with the help of a use case.

Chapter 6 covers the conclusion and future scope of the presented work.

Chapter 2: Column-Oriented Databases

A column-oriented DBMS is a database management system that stores data tables as sections of columns of data rather than as rows of data, like most relational DBMSs [49]. With column store architecture, a DBMS need only read the values of columns required for processing a given query, and can avoid bringing into memory irrelevant attributes. In warehouse environments where typical queries involve aggregates performed over large numbers of data items, a column store has a sizeable performance advantage.

There has been a significant amount of excitement and recent work on column-oriented database systems. These databases show good performance of an order of magnitude better than traditional row-oriented database systems on analytical workloads such as those found in data warehouses, decision support, and business intelligence applications. The reason behind this performance difference is that the column-stores are more I/O efficient for read-only queries since they only have to read from disk those attributes accessed by a query. By denoting one as column-oriented, we are referring to both the ease of expression of a column-oriented structure and the focus on optimizations for column-oriented workloads [48]. This approach is in contrast to row-oriented or row store databases and with correlation databases, which use a value-based storage structure.

2.1 Row-oriented systems

The common solution to the storage problem is to serialize each row of data. As data is inserted into the table, it is assigned an internal ID, the rowid that is used internally in the system to refer to data. In the case of employee records, rowids are independent of the user-assigned empid. The DBMS uses short integers to store rowids, in practice larger numbers, 64-bit or 128-bit, are normally used [50].

Row-based systems are designed to efficiently return data for an entire row, or record, in as few operations as possible. This matches the common use-case where the system is attempting to retrieve information about a particular object, say the contact information for a user in a rolodex system, or product information for an online shopping system. By storing the record's data in a single block on the disk, along with

related records, the system can quickly retrieve records with a minimum of disk operations.

Row-based systems are not efficient at performing operations that apply to the entire data set, as opposed to a specific record. For instance, in order to find all the records in the example table that have salaries between 40,000 and 50,000, the DMBS would have to seek through the entire data set looking for matching records [50]. While the example table shown above will likely fit in a single disk block, a table with even a few hundred rows would not, and multiple disk operations would be needed to retrieve the data and examine it.

To improve the performance of these sorts of operations, most DBMS's support the use of database indexes, which store all the values from a set of columns along with pointers back into the original rowid. As they store only single pieces of data, rather than entire rows, indexes are generally much smaller than the main table stores. By scanning smaller sets of data the number of disk operations is reduced. If the index is heavily used, it can provide dramatic time savings for common operations. However, maintaining indexes adds overhead to the system, especially when new data is written to the database.

There are a number of row-oriented databases that are designed to fit entirely in RAM, an in-memory database. These systems do not depend on disk operations, and have equal-time access to the entire dataset. This eliminates the need for indexes, as it requires the same amount of operations to scan the original data as an index. Such systems are therefore dramatically simpler and smaller, but can only manage databases that will fit in memory.

2.2 Column-oriented systems

A column-oriented database serializes all of the values of a column together, then the values of the next column, and so on. In this layout, any one of the columns more closely matches the structure of an index in a row-based system. This causes confusion about how a column-oriented store is really just a row-store with an index on every column. However, it is the mapping of the data that differs dramatically. In a row-oriented indexed system, the primary key is the rowid that is mapped to indexed data. In the column-oriented system primary key is the data, mapping back to rowids [50]. This may seem subtle, but the difference can be seen in this common

modification to the same store. Other operations, like counting the number of matching records or performing math over a set of data, can be greatly improved though this organization.

Whether or not a column-oriented system will be more efficient in operation depends heavily on the workload being automated. It appears that operations that retrieve data for objects would be slower, requiring numerous disk operations to collect data from multiple columns to build up the record. Whole-row operations are generally rare. In the majority of cases, only a limited subset of data is retrieved. In a rolodex application, operations collecting the first and last names from many rows in order to build a list of contacts are far more common than operations reading the data for any single address. This is even true for writing data into the database, especially if the data tends to be sparse with many optional columns. For this reason, column stores have shown excellent real-world performance in spite of any theoretical disadvantages [50]. In figure 2.1, storage layout of column-oriented and row-oriented systems is shown. It is visible from figure that row-oriented systems serialize the entire row and column-oriented systems serialize the values of a column.

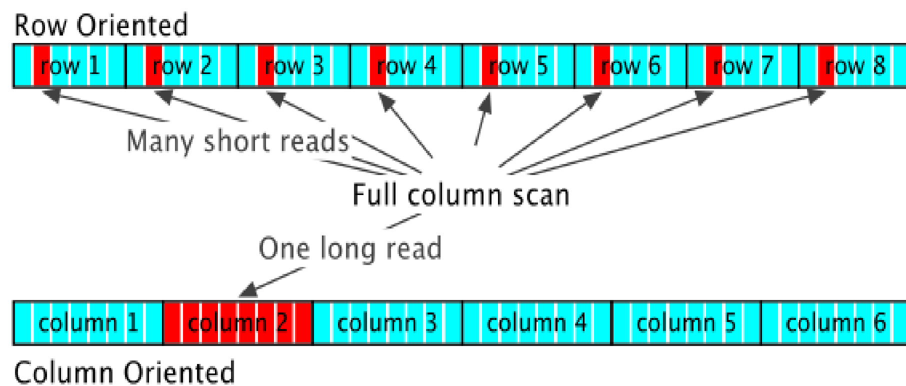


Figure 2.1: Column-oriented vs. row-oriented storage layout [46]

Activities like partitioning, indexing, caching, views, OLAP cubes, and transactional systems such as write ahead logging or multiversion concurrency control all dramatically affect the physical organization of either system. Therefore, online transaction processing (OLTP)-focused RDBMS systems are more row-oriented, while online analytical processing (OLAP)-focused systems are a balance of row-oriented and column-oriented.

2.3 Benefits of Column-oriented databases over Row-oriented Databases

Comparisons between row-oriented and column-oriented data layouts are typically concerned with the efficiency of hard-disk access for a given workload, as seek time is incredibly long compared to the other delays in computers. Sometimes, reading a megabyte of sequentially stored data takes no more time than one random access [51]. Following is few observations which attempt to draw comparison between column- and row-oriented organizations.

1. Column-oriented organizations are more efficient when an aggregate needs to be computed over many rows but only for a smaller subset of all columns of data, because reading that smaller subset of data can be faster than reading all data.
2. Column-oriented organizations are more efficient when new values of a column are supplied for all rows at once, because that column data can be written efficiently and replace old column data without touching any other columns for the rows.
3. Column-oriented organizations are more efficient when many columns of a single row are required at the same time, and when row-size is relatively small, as the entire row can be retrieved with a single disk seek.

In practice, row-oriented storage layouts are preferred for OLTP-like workloads which are more heavily loaded with interactive transactions. Column-oriented storage layouts are preferred for OLAP-like workloads (e.g., data warehouses) which typically involve a smaller number of highly complex queries over all data.

2.4 Available Column-oriented Databases

There are many leading technologies in column-oriented databases but most common are BigTable [53], HBase [40] and Cassandra [54]. HBase is a smaller version of BigTable. Out of these column-oriented databases, HBase and Cassandra are discussed in following section.

2.4.1 HBase

HBase is an open source, non-relational and distributed database [40]. It is developed as part of Apache Hadoop project [59] and runs on top of Hadoop distributed filesystem [40]. It is a column-family database modelled after Google's BigTable [53] and has BigTable like capabilities. It is often described as a sparse, persistent, multidimensional map, which is indexed by rowkey, column key, and timestamp. It is also referred as a key value store and sometimes a database storing versioned maps of maps.

HBase can handle both structured and semi-structured data naturally. It can store unstructured data too, as long as it is not too large [41]. It does not care about types and allows for a dynamic and flexible data model that does not constrain the kind of data to be stored. It does not create any problem in storing an integer in one row and a string in another for the same column. Figure 2.2 shows the internal structure of an HBase database, where the column family is simply an organization unit that groups related columns together.

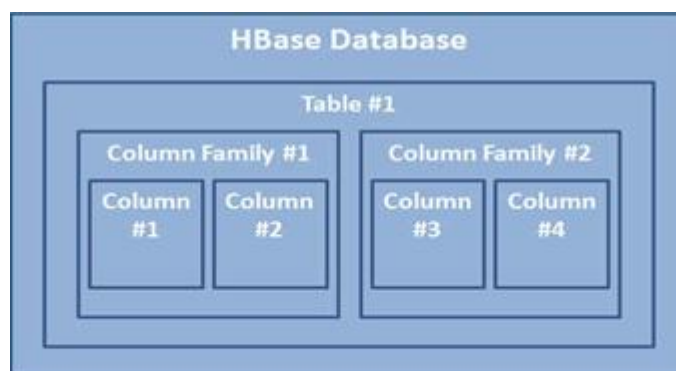


Figure 2.2: HBase Data organisation

2.4.2 Cassandra

Apache Cassandra is an open source distributed database management system [55]. It is an Apache Software Foundation top-level project designed to handle very large amounts of data spread across many commodity servers while providing a highly available service with no single point of failure.

Cassandra provides a structured key-value store with tuneable consistency [54]. Keys map to multiple values, which are grouped into column families. The column families are fixed when a Cassandra database is created, but columns can be

added to a family at any time. Furthermore, columns are added only to specified keys, so different keys can have different numbers of columns in any given family. The values from a column family for each key are stored together. Each key in Cassandra corresponds to a value which is an object. Each key has values as columns, and columns are grouped together into sets called super columns. Also, each super column can be grouped in super column families. In figure 2.3, organization of data in Cassandra has been illustrated. In Cassandra, super column family contains one or more super columns. Each super column contains zero or more columns and is organizationally equivalent to an HBase column family.

Each key identifies a row of a variable number of elements. A table in Cassandra is a distributed multi dimensional map indexed by a key. Furthermore, applications can specify the sort order of columns within a super column or simple column family.

Figure 2.3: Cassandra Data organisation

2.4.3 Comparison between Cassandra and HBase

In this section HBase and Cassandra have been compared in terms of consistency, performance, availability, etc. Although both are popular and widely used databases but in some features one outperforms other. Also this comparison table will help anyone in jeopardy to choose between HBase and Cassandra.

Feature	HBase	Cassandra
CAP Theorem Focus	Consistency, Availability	Availability,Partition-Tolerance
Consistency	Strong	Eventual (Strong is Optional)
Optimized For	Reads	Writes
Main Data Structure	CF, RowKey, Name value	CF, RowKey, Name value

	pair set	pair Set
Dynamic Columns	Possible	Possible
Column Names as Data	Allowed	Allowed
Static Columns	Not allowed	Allowed
RowKey Slices	Possible	Not possible
Cell Versioning	Supported	Not supported
Partitioning Strategy	Ordered partitioning	Random partitioning recommended
Rebalancing	Automatic	Not needed with random Partitioning
Availability	N-Replicas across nodes	N-Replicas across nodes
Data Node Failure	Graceful degradation	Graceful degradation
Data Node Restoration	Same as node addition	Requires node repair admin-action
Data Node Addition	Rebalancing automatic	Rebalancing requires token-assignment adjustment
Data Node Management	Simple (Roll In, Role Out)	Human admin action required
Cluster Admin Nodes	Zookeeper, NameNode, HMaster	All Nodes are Equal
Compression	Support	Support

Table 2.1: Comparison between HBase and Cassandra

For our case study, HBase has been chosen over Cassandra as it is relatively more consistent and mature platform. HBase is not restricted to only traditional HDFS but can change underlying storage depending on the needs. Native map reduce is a concept of simpler schema that can be modified without a cluster restart in HBase that is not possible in Cassandra.

2.5 HBase

HBase is an open source implementation of Bigtable [53]. It is a distributed, persistent, strictly consistent storage system with near-optimal write and provides excellent read performance and makes efficient use of disk space by supporting pluggable compression algorithms that can be selected based on the nature of the data in specific column families.

HBase can run on a cluster of computers instead of a single computer without any major issue. It is possible to scale horizontally by adding more machines to the

cluster. Each node in the cluster provides a bit of storage, a bit of cache, and a bit of computation as well. HBase is also incredibly flexible and distributive. All Nodes are same, so it is simply to replace damaged node with another node.

2.5.1 History

In 2006, Doug Cutting and Mike Cafarella started developing an open source project Nutch [58] for searching the web by crawling websites and indexing them. They implemented that work on a small cluster of machines that requires a lot of manual steps. Nutch was built out of Apache Lucene and became the motivation for the first implementation of Hadoop [59]. From there, Hadoop began to receive lots of attention from Yahoo!, which hired Cutting and others to work on it full time. From there, Hadoop was extracted out of Nutch and eventually became an Apache top-level project.

With Hadoop well underway and the Bigtable paper published, the groundwork existed to implement an open source Bigtable on top of Hadoop [44]. In 2007, Cafarella released code for an experimental, open source Bigtable. He called it HBase. The startup Powerset decided to dedicate Jim Kellerman and Michael Stack to work on this Bigtable analogy as a way of contributing back to the open source community on which it relied.

2.5.2 Data Model

A unique data value in HBase is accessed by way of its coordinates [45]. The complete coordinates to a value are rowkey, column family, column qualifier, and timestamp.

- **Table**:
  HBase organizes data into tables. Table names are Strings and composed of characters that are safe for use in a file system path.
- **Row**:
  Within a table, data is stored according to its row. Rows are identified uniquely by their rowkey. Rowkeys do not have a data type and are always treated as a byte[].

- **Column family**:

  Data within a row is grouped by column family. Column families also impact the physical arrangement of data stored in HBase. For this reason, they must be defined up front and are not easily modified. Every row in a table has the same column families, although a row need not store data in all its families. Column family names are Strings and composed of characters that are safe for use in a file system path.

- **Column qualifier**:

  Data within a column family is addressed via its column qualifier, or column. Column qualifiers need not be specified in advance. Column qualifiers need not be consistent between rows. Like rowkeys, column qualifiers do not have a data type and are always treated as a byte[].

- **Timestamp**:

  Values within a cell are versioned. Versions are identified by their timestamp. When a version is not specified, the current timestamp is used as the basis for the operation.

HBase stores a piece of data in a table based on a 4-d co-ordinate system. The co-ordinates used by HBase in order are rowkey, column family, column qualifier, and timestamp. Figure 2.1 illustrates these coordinates.

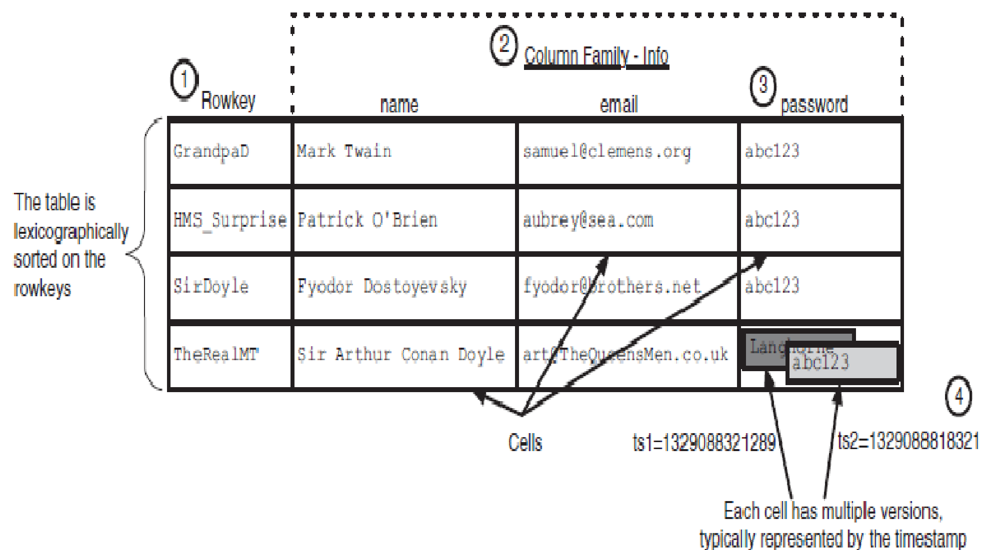


Figure 2.4: The coordinates used to identify data in an HBase table are 1-rowkey, 2-column family, 3-column qualifier, and 4-version [45]

2.5.3 Data Versioning

Data timestamp is a new concept introduced in HBase. There was no database which had feature of storing multiple versions of same data in database. Therefore, once the data had modified, no previous copies of that data was maintained. But every time an operation is performed on a cell, HBase implicitly stores a new version. Creating, modifying, and deleting a cell are all treated identically. Get requests find which version to return based on provided parameters [46]. The version is selected as the final coordinate when accessing a specific cell value. HBase uses the current time in milliseconds when a version is not specified, so the version number is represented as a long. By default, HBase stores only the last three versions. It can be changed umber to an extent. When a cell exceeds the maximum number of versions, the extra records are dropped during the next major compaction. Instead of deleting an entire cell, it is possible to operate on a specific version or versions within that cell. It is possible in HBase to delete a particular version of a cell.

2.5.4 Data Modelling

Mapping from relational database to HBase is not a simple procedure since their underlying data models are indifferent. Relational database modelling consists of three primary concepts:

- Entities: Map to tables.
- Attributes: Map to columns.
- Relationships: Map to foreign-key relationships.

The mapping of these concepts to HBase is an important and essential task and it is explained in the following section.

2.5.4.1 Entities

In both relational databases and column-oriented databases, the default container for an entity is a table, and each row in the table should represent one instance of that entity.

2.5.4.2 Attributes

To map attributes to HBase, first attributes are to be separated into two categories:

Identifying attribute:

This is the attribute that uniquely identifies exactly one instance of an entity (that is, one row). In relational tables, this attribute forms the table's primary key. In HBase, this becomes part of the rowkey, which, is the most important thing to get right while designing HBase tables.

Often an entity is identified by multiple attributes. This maps to the concept of compound keys in relational database systems: for instance, when we define relationships. In the HBase world, the identifying attributes make up the rowkey [45]. The rowkey was formed by concatenating the user IDs of the users that formed the relationship. HBase does not have the concept of compound keys, so both identifying attributes have to be put into the rowkey.

Non-identifying attribute:

Non-identifying attributes are easier to map. They basically map to column qualifiers in HBase. No uniqueness guarantees are required on these attributes. Each key/value carries its entire set of coordinates around with it: rowkey, column family name, column qualifier, and timestamp. If a relational database table with wide rows (dozens or hundreds of columns) is required, it will be easier to store each of those columns as a column in HBase [43]. It will take less disk space, but it has downsides: the values in the row are now opaque. When the storage footprint is important and the access patterns always involve reading entire rows, this approach is good.

2.5.4.3 Relationships

Logical relational models use three main types of relationships: one-to-one, one-to-many and many-to-many. Relational databases model the second directly as foreign keys (whether explicitly enforced by the database as constraints, or implicitly referenced by your application as join columns in queries) and the latter as junction tables (additional tables where each row represents one instance of a relationship between the two main tables).

2.5.5 Nested entities

One thing that is significantly different about HBase is column also known as column qualifiers are not predefined at design time.

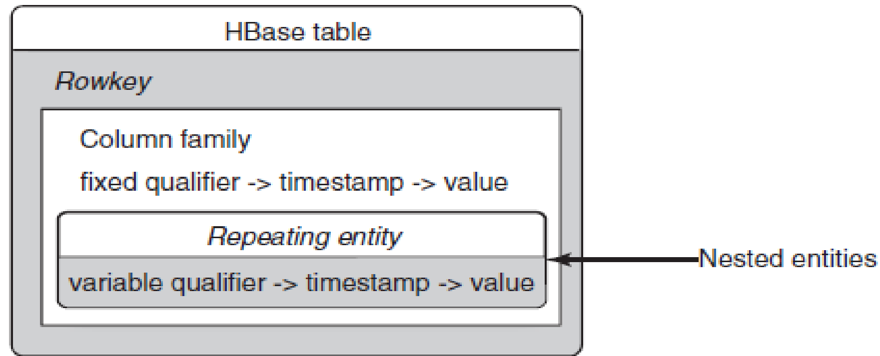


Figure 2.5: Nesting entities in an HBase table

Being a flexible schema row, this represents the ability to nest another entity inside the row of a parent or primary entity. If tables exist in a parent-child, master-detail, or other strict one-to-many relationship, it is possible to model it in HBase as a single row. The rowkey will correspond to the parent entity [42]. The nested values will contain the children, where each child entity gets one column qualifier into which their identifying attributes are stored, with the remainder of the non-identifying attributes stashed into the value. The real HBase row defines the parent record and records of the child entity are stored as individual columns.

There are some limitations of nested entities also. First, this technique only works to one level deep that is the nested entities cannot have nested entities. It is still possible to have multiple different nested child entities in a single parent, and the column qualifier is their identifying attributes. Second, it is not as efficient to access an individual value stored as a nested column qualifier inside a row, as compared to accessing a row in another table.

2.5.6 Business Use-cases

The adoption of HBase has grown rapidly over the last couple of years. This has been resultant of the system becoming more reliable and performant, due to large part of engineering effort invested by the various companies backing and using HBase.

1. Canonical web-search

Searching for documents containing particular terms requires looking up indexes that map terms to the documents that contain them [44]. HBase provides storage for the

corpus of documents. Bigtable supports row-level access so crawlers can insert and update documents individually. The search index can be generated efficiently directly against Bigtable. Individual document results can be retrieved directly. Support for all these access patterns was the key in influencing the design of HBase.

2. Capturing incremental data

Data often added to an existing data store for further usage, such as analytics, processing, and serving. HBase acts as a data store that captures incremental data coming in from various data sources. Facebook uses the counters in HBase to count the number of times people like a particular page. Content creators and page owners can get near real-time metrics about how many users like their pages. This allows them to make more informed decisions about what content to generate. Facebook built a system called Facebook Insights, which needs to be backed by a scalable storage system. The company looked at various options, including RDBMS, in-memory counters, and Cassandra, before settling on HBase [40]. This way, Facebook can scale horizontally and provide the service to millions of users as well as use its existing experience in running large-scale HBase clusters. The system handles tens of billions of events per day and records hundreds of metrics.

3. Content serving

One of the major use cases of databases traditionally has been that of serving content to users. Applications that are toward serving different types of content are backed by databases of all shapes, sizes, and colors [44]. Users consume and generate a lot of content. HBase is being used to back applications that allow a large number of users interacting with them to either consume or generate content.

A content management system (CMS) allows for storing and serving content, as well as managing everything from a central location. More users and more content being generated translate into a requirement for a more scalable CMS solution. Lily, a scalable CMS, uses HBase as its back end, along with other open source frameworks.

4. Information exchange

The world is becoming more connected by the day, with all sorts of social networks coming up. One of the key aspects of these social networks is the fact that users can interact using them. Sometimes these interactions are in groups other times; the

interaction is between two individuals [40]. Think of hundreds of millions of people having conversations over these networks. They are not happy with just the ability to communicate with people far away; they also want to look at the history of all their communication with others.

One such use case that is often publicly discussed and is probably a big driver of HBase development is Facebook messages. All messages that users write or read are stored in HBase. The system supporting Facebook messages needs to deliver high write throughput, extremely large tables, and strong consistency within a datacenter.

Chapter 3: Literature Survey on Schema Mapping

Schema Mappings are specifications that model a relationship between two data schemas. It transforms a source database instance into an instance that obeys a target schema [22]. It is typically formalized as a triple (S, T, Σ) where S is a source schema, T is a target schema, and Σ is a set of dependencies and constraints that specify the relationship between the source schema and the target schema. A schema is a collection of named objects which provides a logical classification of objects in the database. Some of the objects that a schema may contain include tables, views, aliases, indexes, triggers, and structured types.

Schema Mappings are key elements in any system that requires the interaction of heterogeneous data and applications. Such interaction usually involves databases that have been independently developed and that store the data of the common domain under different representations; that is, the involved databases have different schemas. In order to make the interaction possible, schema mappings are required to indicate how the data stored in each database relates to the data stored in the other databases [20]. This problem, known as information integration, has been recognized as a challenge faced by all major organizations, including enterprises and governments.

Two well-known approaches to information integration are data exchange and data integration. In the data exchange approach, data stored in multiple heterogeneous sources are extracted, restructured into a unified format, and finally materialized in a target schema. In particular, the data exchange problem focuses on moving data from a source database into a target database, and a mapping is needed to specify how the source data is to be translated in terms of the target schema [21]. In data integration, several local databases are to be queried by users through a single, integrated global schema; mappings are required to determine how the queries that users pose on the global schema are to be reformulated in terms of the local schemas.

3.1 Schema Mapping Models

Due to the importance of the schema mapping, a handful of mapping design systems has been developed. These systems are categorized into two basic models, match-driven model and example-driven model.

3.1.1 Match-driven Model

Match-driven methodology consists of two phases. Specifically, first a visual specification of the relationship between elements of the source and target schemas is solicited from the user (perhaps with the help of a schema matching module) and then a schema mapping is derived from the visual specification [23]. However, several pair wise logically in equivalent schema mappings may be consistent with a visual specification, and differing schema mappings may be produced from the same visual specification, depending on which system is used. Thus, the user is still required to understand the formal specification in order to understand the semantics of the derived schema mapping. InfoSphere Data Architect, BizTalk Mapper, Altova MapForce, and Stylus Studio systems are based on Match-driven Methodology.

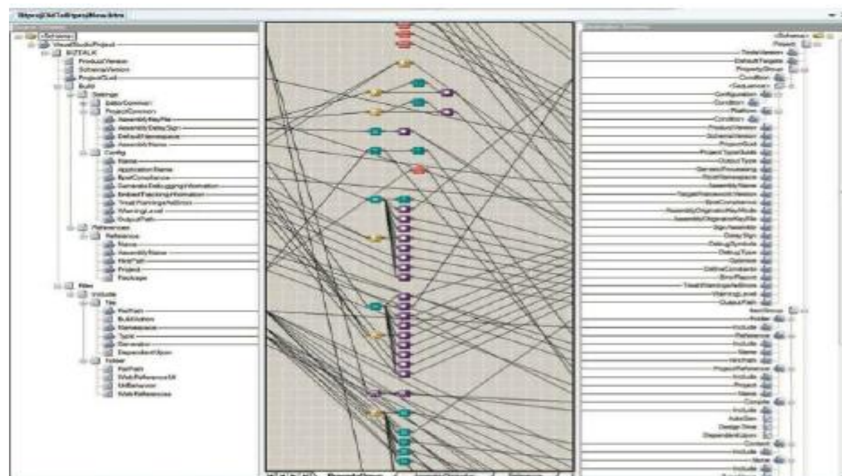


Figure 3.1: Visual specification of value correspondences [22]

Unfortunately, this traditional match-driven model is not suitable for many modern schema mapping tasks. The user must either build attribute-level matches from scratch, or else check an automatically-generated set of matches. An implicit assumption made by these systems is that the user has detailed knowledge of both the source and target schemas [23]. For traditional schema mapping tasks that involve a

sophisticated administrator and a single high-value target database, this assumption makes sense. But modern mapping scenarios feature relatively unsophisticated users and a multiplicity of tasks. For these less-technical users who perform a large number of mappings, the laborious match-driven process can be a heavy burden.

3.1.2 Example-driven Model

Example-driven develops a schema mapping by using data example in an interactive scenario. Each data example represents a partial specification of the semantics of the desired schema mapping. Furthermore, the user stipulates that, for each data example, a particular target instance is a universal solution for source instance with respect to the desired schema mapping [24]. The system responds by generating a GLAV schema mapping that fits the data examples or by reporting that no such GLAV schema mapping exists. Here, we say that a schema mapping M fits a set E of data examples (I,J), if for every data example, the target instance J is a universal solution of the source instance I with respect to the M. The user is then allowed to modify the data examples in E and provide the system with another finite set E0 of data examples. After this, the system tests whether or not there is a GLAV schema mapping that fits E0 and, as before, returns such a GLAV schema mapping, if one exists, or reports that none exists, otherwise [25]. The cycle of generating schema mappings and modifying data examples can be repeated until the user is satisfied with the schema mapping obtained in this way. This process is called interactive refinement of a schema mapping via data examples.

3.2 Popular schema mapping tools

There are many known schema mapping tools for relational and xml databases. In the following section, few major schema mapping are discussed. *

3.2.1 Clio

Clio is a schema mapping prototype developed jointly between IBM Almaden Research Center and the University of Toronto in 1999 [26]. Clio automatically creates mappings using two schemas and a set of value mappings between their atomic elements. Clio builds an internal representation, capable of representing relational and XML schemas. Depending on the kind of source and target schemas,

Clio can render queries that convert source data into target data in a number of languages. Clio generates a set of mappings that would map all source data. A user can then choose among these mappings. If it only wishes to map a subset of the source data, she can do so by selecting an appropriate subset of the mappings Clio generates [27]. Clio can generate code that, given a source instance, will produce an instance of the target schema that satisfies the mapping and that represents the source data as accurately as possible.

3.2.2 Clip

Clip is a new schema mapping language developed at Politecnico di Milano [28]. Users of Clip enter mappings by drawing lines across schema elements and by annotating some of the lines. Clip then produces the queries that implement the mapping. The main difference introduced by Clip relative to its predecessors, and most specifically to Clio that greatly influenced Clip's design, is the introduction of structural mappings in addition to value mappings. This gives users greater control over the produced transformations. Clip mappings are less dependent on hidden automatisms, whose assumptions may fail in capturing the intended semantics.

Given a source and a target schema, Clip identifies source and target tableaux. After the source and target tableaux are computed, Clip creates a matrix source vs. target tableaux. Each entry in this matrix relates a source with a target tableaux and is called a mapping skeleton [29]. For each value mapping entered by the user, it matches the source and target end-points of the value mappings against all the mapping skeletons and marks as active those skeletons encompassing some value mappings. Each active skeleton that is not implied or subsumed by others emits a logical mapping.

3.2.3 Eirene

Eirene is a novel system for designing and refining schema mappings interactively via data examples. The intended users of our system are mapping designers who wish to design a schema mapping over a pair of source and target relational schemas. It is tailored for schema mappings specified by GLAV (Global-and-Local-As-View) [30] constraints also known as source-to-target tuple generating dependencies. Such schema mappings are called GLAV schema mappings.

The interaction between the user and the system begins with the user providing an initial finite set E of data examples, where a data example is a pair (I,J) consisting of a source instance and a target instance that conform to a source and target relational schema. Intuitively, each data example in E provides a partial specification of the semantics of the desired schema mapping [31]. The system processes all the input data examples and responds by generating a schema mapping that fits all the data examples in E or if no schema mapping exists it reports to the user.

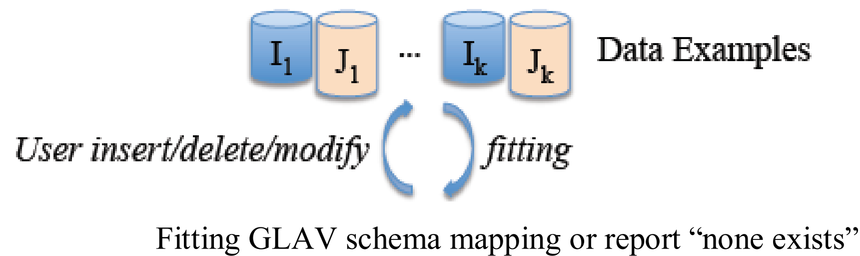


Figure 3.2: Workflow for interactive design and refinement of schema mappings via data examples [34]

A schema mapping M fits a set S of data examples if for every data example (I,J), the target instance J is a universal solution of the source instance I with respect to M. Now the user can modify the existing data examples in S or provide the system with another finite set S1 of data examples [33]. The system again checks whether or not there is a schema mapping that fits S1 and, as before, returns such a GLAV schema mapping, if one exists, or reports that none exists, otherwise. The iteration of collecting and modifying example data and generating schema mappings continues until the user is satisfied with the schema mapping. Workflow of the system is depicted in the following figure [34].

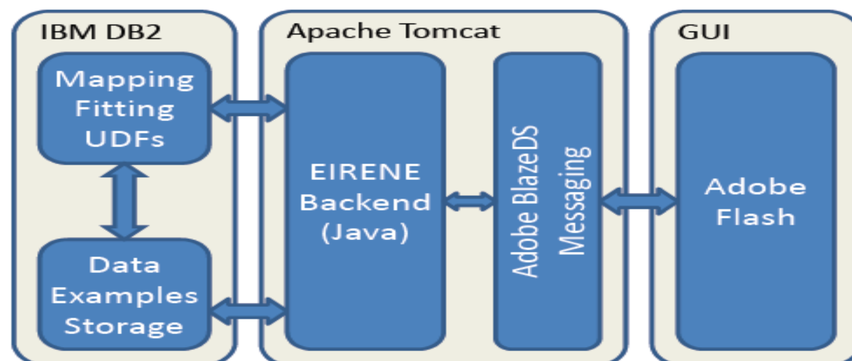


Figure 3.3: Eirene: System Architecture [34]

3.2.4 MWeaver

All the tools studied till now, provide powerful functions to generate schema mappings, but they usually require an in-depth understanding of the semantics of multiple schemas and their correspondences. Thus these are not suitable for users who are technically unsophisticated or when a large number of mappings must be performed.

MWeaver is a schema mapping system that provides sample-driven schema mapping. The key idea behind this approach is to allow the user to implicitly specify schema mappings by providing sample data of the target database [35]. User does not need to have complete knowledge of source schema but in sample-driven, match-driven approach requires in-depth of both source and target schema. Figure 3.4 depicts a high-level comparison between the traditional match-driven approach and the sample-driven approach.

The system automatically selects the mappings that transform the source database into given sample data (partial target instance). If the user provides enough information, the system can determine a single best mapping. The process is iterative and similar to Eirene. As the user types more information from the target database, the system provides increasingly better estimates of the correct mapping.

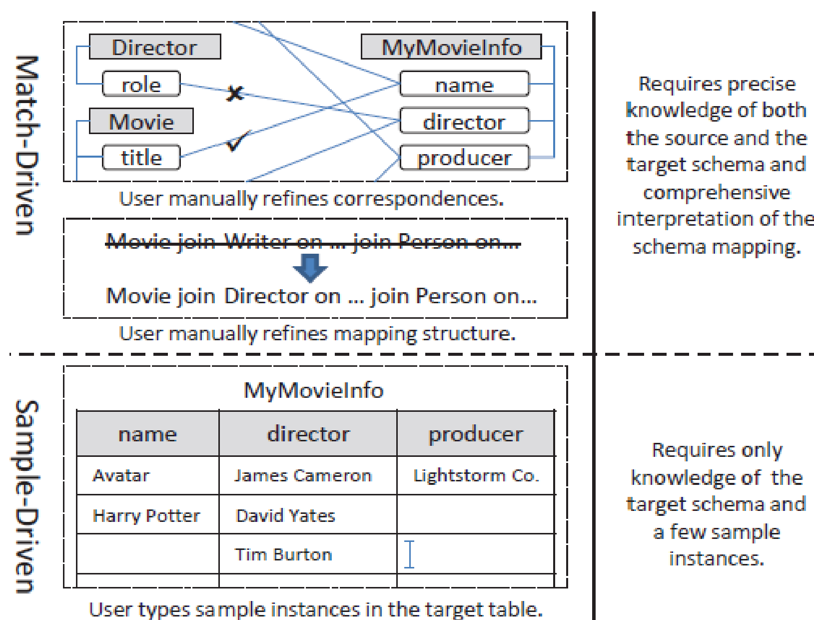


Figure 3.4: A Comparison between the Match-driven Approach and the Sample-driven Approach [35]

The primary user interface component of MWeaver is a spreadsheet that conforms to the target schema. There is an input spreadsheet on the left of Figure 3.5 that shows an input spreadsheet in which the user fills the data. The user may adjust the input spreadsheet by adding/dropping/renaming columns to meet the model of the target. The bar directly under the logo provides information about the current mapping generation status [35]. The user can select one of the mappings from a mapping list, which visualizes each mapping with details.

Afterwards, the user may continue to provide sample instances. Whenever one cell is updated, MWeaver uses all the samples from to prune the set of mappings. This process is called Sample Pruning. Finally, the interaction terminates when there is only one mapping left. In other words, as the number of candidate mappings decreases, the average mapping quality increases with respect to the number of user-input sample.

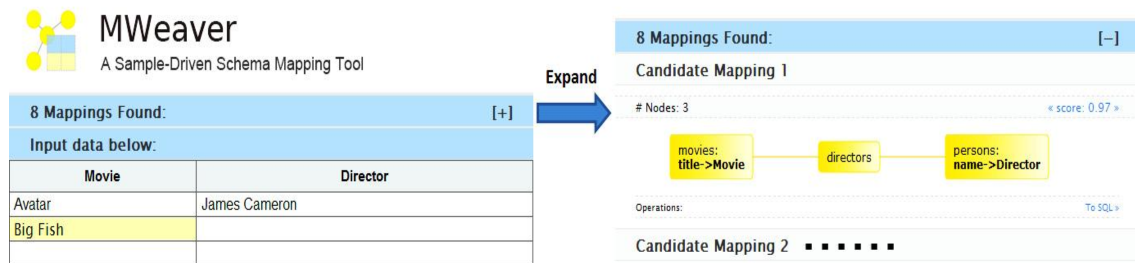


Figure 3.5: A screenshot of MWEAVER. Left: the input spreadsheet. Right: the expanded list of candidate mappings [35]

The sample-driven approach reduces the user's cognitive burden in two ways. First, the user no longer needs to explicitly understand the source database schema or the mapping. She simply types in sample instances until the system converges to a single proposal. Second, the operations that the user performs are common and require no special training. In contrast, current tools are applications designed for trained DBAs, and require users to decide whether individual attribute-level matches are correct. MWeaver system enables users to perform practical mapping tasks in about 1/5th the time needed by the known tools.

3.3 Representing Schema Mappings

Schema mapping is a high-level specification that describes the relationship between two database schemas. Schema mappings constitute essential building blocks of data integration, data exchange and peer-to-peer data sharing systems. Global-and-local-

as-view (GLAV) is one of the approaches for specifying the schema mappings. Tableaux are used for expressing queries and functional dependencies on a single database.

3.3.1 GLAV

Schema Mappings constitute the building blocks of data integration system, data exchange system and P2P data sharing system. A schema mapping describes the relationship between two database schemas at high level [46]. There are three approaches for specifying the schema mappings: local-as-view (LAV), global-as view (GAV), and global-and-local-as-view (GLAV).

GAV approach is global-schema-centric (or target-schema-centric in case of data exchange) and specifies the mappings by a set of assertions of the form

$$\forall x(s(x) \rightarrow g(x))$$

where g is an element of global schema (target schema) G and s is a query over source schema S. Relations in G are views and queries that are expressed over the views. Queries can simply be evaluated over the data satisfying the global relations. LAV approach is source-centric and specifies the mappings by a set of assertions of the form

$$\forall x\ (s(x) \rightarrow \alpha(x))$$

where s is an element of S and α is a query over G. GLAV is a mixed approach and specifies the mappings by a set of assertions of the form

$$\forall x(\exists\ ys(x,y) \rightarrow z\exists\ g(x,z))$$

where g is a query over G and s is a query over S. Note that all of GAV, LAV and GLAV mappings are represented by First Order Logic (FOL) statements and queries are important part of the mappings [46].

3.3.2 Tableaux

A tableau is a set of schema elements or attributes that are semantically related; elements are related, for instance, when they are siblings under the same repeating element (the columns of a table in a relational schema), or when the containing repeating elements are related via a parent-child relationship. Intra-schema constraints are also used to define tableaux by chasing existing tableaux over the constraints [36].

Tableaux are a way of describing all the basic concepts and relationships that exist in a schema. There is a set of tableaux for the source schema and a set of tableaux for the target schema. Each tableau is primarily an encoding of one concept of a schema. In addition, each tableau includes all related concepts, that is, concepts that must exist together according to the referential constraints of the schema or the parent-child relationships in the schema.

Essentially, a schema tableau is constructed by taking each relation symbol in the schema and chasing it with all the referential constraints that apply. The result of such chase is a tableau that incorporates a set of relations that is closed under referential constraints, together with the join conditions that relate those relations. For each relation symbol in the schema, there is one schema tableau.

3.3.2.1 History

The chase procedure was initially developed for testing logical implication between sets of dependencies, for determining equivalence of database instances known to satisfy a given set of dependencies and for determining query equivalence under database constrains. The chase has been used to compute representative target solutions in data exchange [37]. Intuitively, the data exchange problem consists of transforming a source database into a target database, according to a set of source to target dependencies describing the mapping between the source and the target. The set of dependencies may also include target dependencies, that is, constraints over for the target database.

The source and the target schemas are considered to be distinct. Given a source instance X and a set ∑ of dependencies, an instance Y over the target schema is said to be a solution if X U Y satisfies all dependencies in ∑. One of the most important representations for this set of solutions was introduced by Fagin et al. [6]. They considered the finite tableau obtained by chasing the initial instance with the dependencies. Such a tableau, if exists, was called a universal solution.

In particular, the universal solution can be used to compute certain (unions of) conjunctive queries over the target instance. The universal solution is not a good candidate for materialization in case of non-monotonic queries, or even conjunctive queries with inequalities. The first closed world approach in data exchange was introduced by Libkin [38] and extended by Hernich and Schweikardt [39].

3.3.2.2 Relational tableaux

*Relational instances***:** Let Cons be a countably infinite set of constants, usually denoted a, b, c,...., possibly subscripted. From the domain Cons and a finite set R of relation names, build up a Herbrand structure consisting of all expressions of the form R(a1, a2, . . . , ak), where R is a "k-ary" relation name from R, and the ai's are values in Cons. Such expressions are called ground atoms, or ground tuples. A database instance I is then simply a finite set of ground tuples, e.g. {R1(a1, a3),R2(a2, a3, a4)}. It is denoted as the set of constants occurring in an instance I with dom(I) [37].

Let Vars be a countably infinite set, disjoint from the set of constants. Elements in Vars are called variables, and are denoted X, Y,Z, . . ., possibly subscripted. It can then also allow non-ground atoms, i.e. expressions of the form R(a,X, b, . . . ,X, Y). A tableau T is a finite set of atoms. A non-ground atom represents a sentence where the variables are existentially quantified. The set of variables and constants in a tableau is denoted dom(T).

Let T and U be tableaux. A mapping h from dom(T) to dom(U), that is the identity on Cons, extended component-wise, is called a homomorphism from T to U if $h(T) \subseteq U$. If there is an injective homomorphism h such that h(T) = U, and the inverse of h is a homomorphism from U to T, we say that T and U are isomorphic. An endomorphism on a tableau T is a mapping on dom(T) such that $h(T) \subseteq T$. Finally, a valuation is a homomorphism whose image is contained in Cons.

3.3.2.3 **Tuple generating dependencies**

A tuple generating dependency (tgd) is a first order formula of the form

$$\forall x : \alpha(x,y) \rightarrow \exists z\ \beta(x, z),$$

where α is a conjunction of atoms, β is a single atom, and x, y and z are sequences of variables. The variables occurring in dependencies are disjoint from all variables occurring in any tableaux under consideration [37].

The variables occurring in dependencies are disjoint from all variables occurring in any tableaux under consideration. To emphasize this, we use lowercase letters for variables in dependencies. When α is the antecedent of a tgd, we shall sometimes conveniently regard it as a tableau, and refer to it as the body of the tgd. Similarly we refer to β as the head of the tgd. If there are no existentially quantified

variables the dependency is said to be full, otherwise it is embedded. Frequently, we omit the universal quantifiers in tgd formulae. Also, when the variables are not relevant in the context, we denote a tgd $\forall x : \alpha(x,y) \rightarrow \exists z\ \beta(x, z)$ simply as $\alpha \rightarrow \beta$.

Chapter 4: Problem Statement and Motivation

Web and mobile applications have been transformed according to the need of today's world. A modern web application is designed to support large number of concurrent users by spreading load across a collection of application servers. These systems require interaction of heterogeneous data and applications. This interaction involves databases that have been independently developed and store the data of the common domain under different representations. The interactions are made possible using schema mappings and transformation. This problem is known as information integration. It has been recognized as a major problem faced by all major organizations, including enterprises and governments.

With the development of the Internet and cloud computing, there is need of databases that can store and process big data effectively, be able to satisfy demand for high performance when reading and writing, so the traditional relational database is facing many challenges. Especially in large scale and high concurrency applications, such as search engines, data warehouses and social networking services, using the relational database to store and query dynamic user data has appeared to be inadequate. To meet the reliability and scaling needs, several storage technologies have been developed to replace relational database for structured and semi-structured data and these new technologies are called NoSQL databases. As a result, relational database of existing applications must be transformed into these NoSQL databases so that they can run on such platforms.

Column-oriented databases are the only class of NoSQL databases that are appropriate for OLAP-like workloads (e.g., data warehouses and customer relationship management systems) which typically involve a smaller number of highly complex queries over all data and for storage size optimization which involves compression of uniform column data.

Most of the transformation systems that have been developed till now, generate mappings between any combinations of relational and XML schemas. However, to the best of our knowledge, no work has been done to transform a relational database into a column-oriented database. Therefore, an algorithm for transforming a relational database (PostgreSQL) to a column-oriented database (HBase) using schema mapping has been proposed in this thesis.

Chapter 5: Design and Implementation

In this section, a novel solution has been proposed to represent HBase database by taking an example database of PostgreSQL database and to transform the PostgreSQL into HBase using tableaux and mapping queries. Based on the data model and features of HBase, a transformation algorithm is presented that might be useful for transforming other NoSQL databases in future.

HBase is an open source implementation of BigTable. It is sparse, persistent and multidimensional map, which is indexed by rowkey, column key, and timestamp. It uses a data model very similar to that of BigTable. It is also referred as a Column-oriented store and sometimes a database storing versioned maps of maps. A data row in HBase has a sortable row key and an arbitrary number of columns, which are further grouped into sets called column families. Each data cell in HBase can contain multiple versions of the same data which are indexed by timestamp. If not specified, data with the latest timestamp will be read and written by default.

To differentiate from PostgreSQL databases whose data is usually represented as a table form, we represent an HBase table here as a sorted map. Consequently, a data cell in the table can be viewed as a key value pair where the key is a combination of row key, column and timestamp; and the value is an uninterpreted array of bytes.

```
RowKey[RK] {
    Column Family CF1 {
        Column P:
            ts1 Value1
            ts2 Value2
            ts3 Value3
        Column Q:
            ts4 Value3
        }
    Column Family CF2 {
        Column E:
            ts5 Value4
            ts6 Value5
```

ts7 Value6

}

}

Figure 5.1: HBase data model

5.1 Example database overview

A use case has been taken to make it better understand, how the schema transformation is presented in the algorithm. A Post-Comment system has been considered in the use case. In which, a user can post on site and comment under any post or comment. Post can be written by any user at any time. Tags are used to tag the posts. Therefore, the system has four tables. The four tables are only real entity in the entire example system and they are interlinked with each other using foreign keys.

Four Tables are User, Post, Comment and Tag. User table attributes are uid, username, realname, email, homeurl and pswd. Uid is the id of the user and it is unique. It is the primary key of the user table. All the other attributes can be duplicate. Each user in the system must have a valid account. The user can login in the system using its username and password. Homeurl is the url of its home page.

Attribute of Post table are pid, uid, time, title, body and primaryurl. Pid is the primary key of the table. Any user can post in the system without having valid userid. Uid is the id of the user, who posted the article. If user is an outsider, uid will be Null.

Attributes of Comment table are cid, parentid, postid, uid, title, comment, score, and descriptor. Primary key of the table is cid. Only valid users are allowed to comment under the post. Nested comments are allowed. User can comment under a comment. Score tells the usefulness of the material. Posted is the post id under which this comment is posted and parentid is the cid of the comment under which it is written. Comment table is joined with post table using postid and to itself using parentid.

Attributes of Tag table are tagid, name and pid. Primary key is tagid. To categorize the posts in the system tags are assigned. Tags can be used to narrow the search and is joined with post table using pid.

5.1.1 Schema of example database:

User(<u>uid</u>, username, realname, email, homeurl, pswd)

Post(<u>pid</u>, uid, time, title, body, primaryurl)

Comment(<u>cid</u>, parentid, postid, uid, title, comment, score, descriptor)

Tag(<u>tagid</u>, name, pid)

Primary keys are underlined and foreign keys are shown using arrows in the below diagram of example schema.

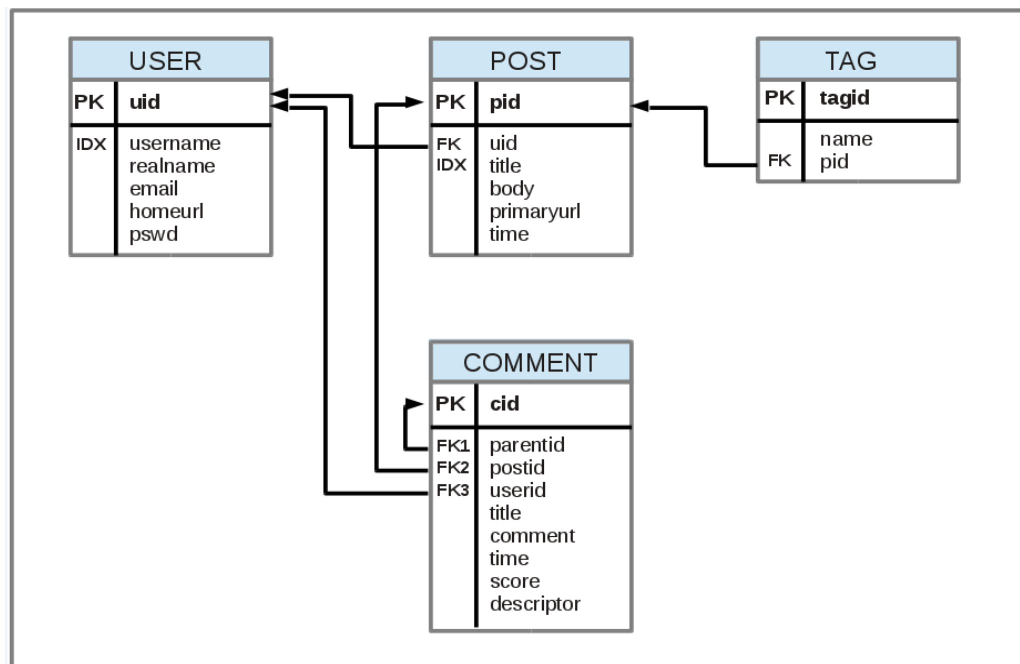


Figure 5.2: Schema of source database

5.2 Schema Transformation and Mapping

The transformation of relational database into HBase is divided into two steps. First step takes the relational database as input and transform it using features and data model of HBase. Once the HBase schema is created in first step, it is then mapped to the relational database using nested mapping queries. A heuristic approach is used to transform the data from source to target database.

5.2.1 Transformation of schema

In this phase, target schema is derived by applying some simple rule on source schema. As the source schema is a relational schema, it is already in normalized form.

There is a unique rowkey for each table in HBase. Each row in the table has a unique rowkey value that separates it from other rows. With the help of these rowkeys, one table is connected to another.

Figure 5.3: Transformation overview

Column-family is a new concept in HBase. A column family must be created before data can be stored in it and does not change during any operation. Same column family data is stored together on the file system while data might be distributed on several machines for different column families. On the other side a table may have any number of columns and many columns can be added into a family dynamically. Therefore, it is possible to have different rows of a table having the same column families but completely different columns.

Algorithm: SchemaTransformation

Input: Source schema as **S**
Output: Target schema as **T**

```
1. String table[] := get_table_names(S)
2. for all i € {1 . . tab.len} do
3.          String col[] := get_column_names(table[i])
4.          String PK := get_primary_key(table[i])
5.          String HBaseTable := create_table( Tab[i], Col, Rowkey:PK)
6.          String CF[] := group_correlated_columns(Col)
7.          String EK[][] := get_exported_keys(tab[i])
8.          for all j € {1 . .  EK.len} do
9.                   int unique := check_uniqueness(EK[1][j] , EK[2][j])
10.                  String  EK_PK = get_primary_key(EK[1][j])
11.                  if  (!unique)
12.                       add_column( HBaseTable, Rowkey:EK_PK)
13.                  else
14.                       add_column_family(HBaseTable, Rowkey:EK_PK)
15.         end for
16. end for
17. String main_tables[]:= find_main_tables(S)
18. String attached_tables[]:= find_attached_tables(s)
19. T:= mergeTables(main_tables, attached _tables)
20. return T
```

An algorithm has been proposed for schema transformation from relational database to column-oriented database in the above section. Detailed explanation of all the functions used in the algorithm along with parameters received and returned is available as an appendix.

The proposed algorithm can be basically divided into four main steps which are explained below.

1. **Extract all the tables** from source database and create same tables in Hbase.

    In the algorithm, *get_table_names()* takes source schema and return all the tables of the source schema. All table names are stored in an array of string named Tab. For each table, the column names are extracted using *get_column_names()* function and the primary key is get using *get_primary_key()*. Now the schema extraction is complete but no table is created till now. So tables are created in target database T with same names as in table array and column names as in col array. Primary key of the table in source schema will be taken as the rowkey of the same table in target schema.

2. **Create new column families** by grouping correlated columns.

    No concept of column family is introduced in first step. So the next will be to create column families and group correlated data. The columns that are accessed together or represent the same or related information are grouped together in a column family. The *group_correlated_columns()* function takes col (array of column names) and correlate the columns and create new column families by putting correlated columns in a single column family. It returns name of the column families.

3. Add rowkey information of one table in another table by creating either new column or column family for **joining the tables**.

    There are no join operations possible in column-oriented databases. Therefore, additional information is stored in tables to facilitate easy access of one table by another. The *get_exported_keys()* function takes a table and returns array of tables and their foreign keys those are connected to the input table with its primary key. Exported keys are required to find the relationships between tables of source database. If 1:1 relationship exists, then only a new column is added to the primary table else a new column family is added to both the tables. If foreign key attribute contains duplicate values, then the relationship is 1: n else it is 1:1 relationship.

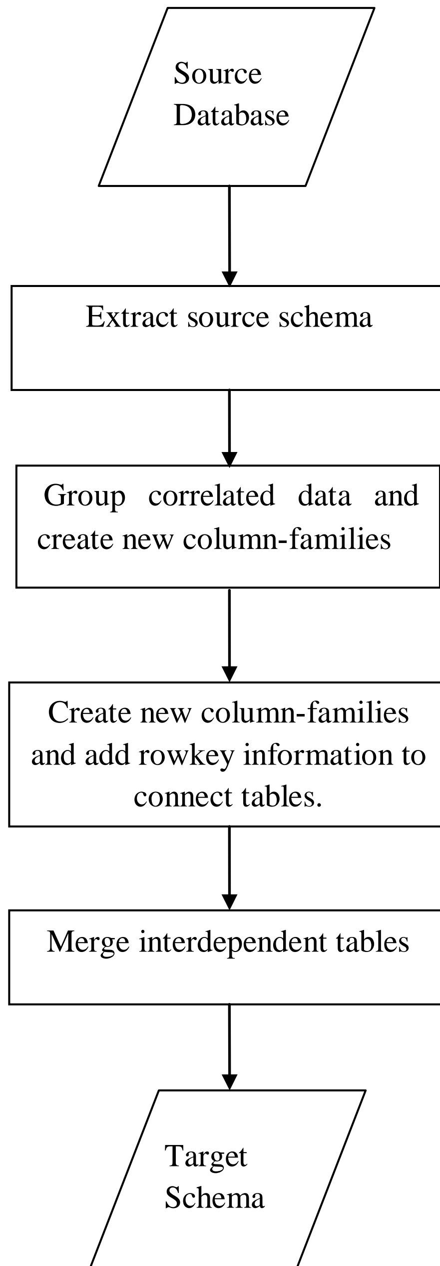


Figure 5.4: Flowchart of proposed schema transformation algorithm

4. **Merge dependent tables** to reduce duplicate information.

The source tables can be divided into main and attached tables. The main tables are the real independent entities whereas attached tables are dependent entities. The *find_main_tables()* and *find_attached_tables()* functions allow user to decide which tables to be assigned as main and attached. Then attached tables are merged into main tables by creating new column families. This was the last step in the algorithm and after merging main and attached tables, the tables left together make

target database. Figure 5.5 shows the flowchart of the schema transformation algorithm.

The proposed algorithm is a generic algorithm. To explain the working of algorithm we have elaborated the above given four steps of the transformation by taking the use case of Post-comment system.

1. Extract source schema

In the first step each table in the source database is converted into HBase table with primary key changed to rowkey. All the columns of a table are placed in one column family of the target table. We extract table names and attributes of the tables and create tables with same table names and group all the attributes in a single column family.

Source database contains four tables and these tables have unique primary keys. After first step four target tables are created in HBase and their rowkeys are same as primary keys of source tables. These tables are listed below:

Table name	Rowkey
User	Uid
Post	Pid
Comment	Cid
Tag	Tagid

Table 5.1:Target table and respective rowkeys

2. Create new column families by grouping correlated data

Now the question arises, what is the purpose of column families if we put all the columns in one column family? It is answered in the next step, in which we group all related columns together and put them into different column families as all these columns tend to be read and written together. New column families are created for each table on the basis of correlation of columns. Once new column families are decided, all the columns are group in their respective column families.

In User table two column families are created. The information about the user is stored in *Info* column family and its login information is stored in *Credential* column family. It is clear from the arrangement of the columns in column families that login data is kept separate as it will only be needed at the time of login. Columns in Post table are also grouped in two column families: *Info* and *text*. *Info* contains the information or meta data about the post and text contains the title and post text. In the same way Comment table is divided into three column families: *info, text* and *score*. The resultant schema after second step is shown in figure 5.6 shown.

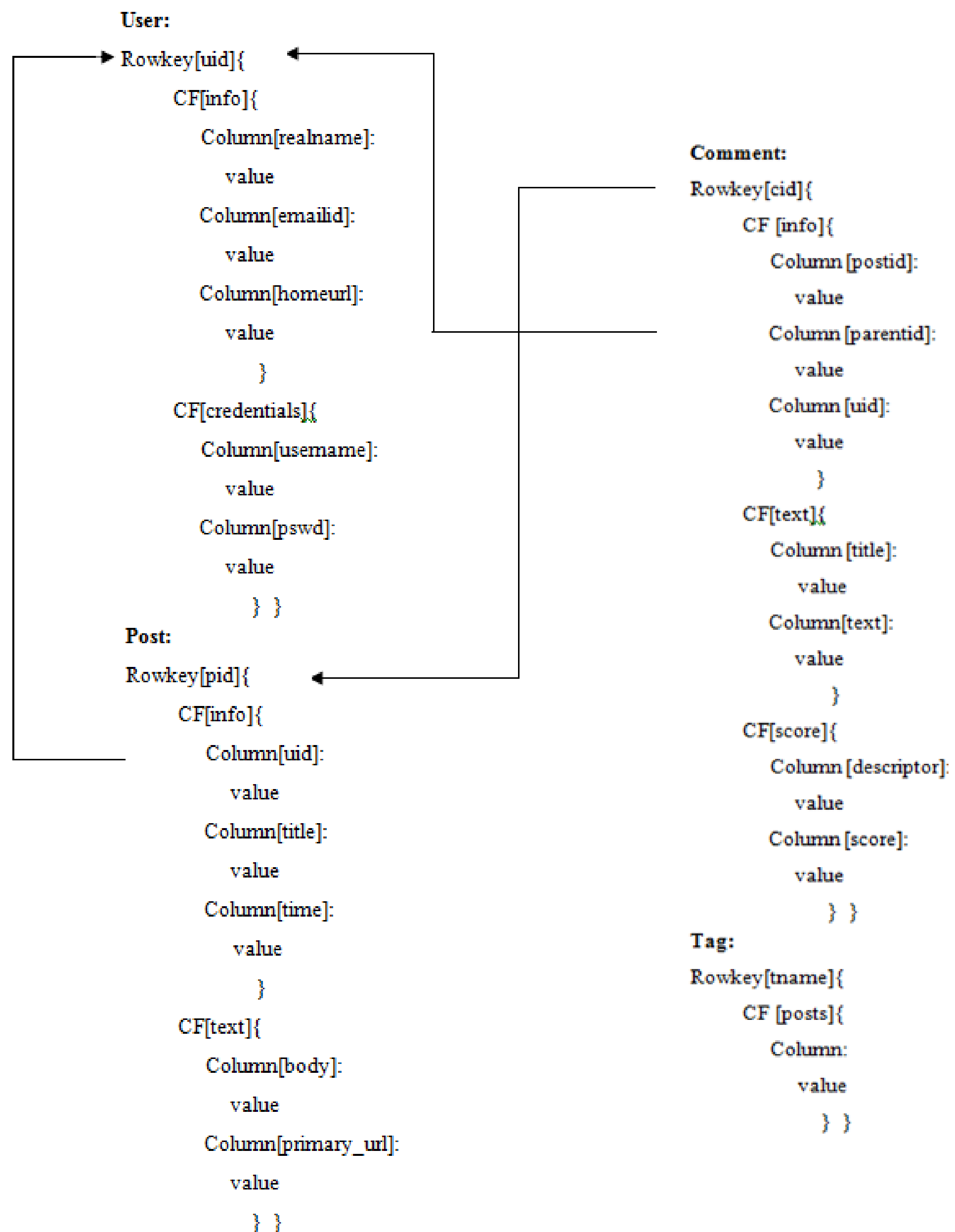


Figure 5.5: Target schema transformation after rule 2

3. Add rowkey information to create relationships between objects

There is no foreign keys concept in HBase. For connecting two tables, rowkey information of the referred row to the referring table is to be added. Without knowing rowkey, one row cannot refer other row. HBase does not support joins but maintains links through rowkeys. Therefore, it requires less time to reach to a particular data and perform operations. There are three types of relationships in relational database and we have to consider all of them in HBase too.

One-to-one relationship

It is easy to handle in HBase as foreign key can be placed on both sides of the relationship by creating new column. This column can be grouped with any other column without any problem. For example, uid column in Post table contains the rowkey of user table and it is treated like an ordinary column and it is grouped in *info* column family.

One-to-many relationship

Due to normal form, it is not possible to have multiple values for an attribute in relation database. But in HBase, multiple values are allowed by grouping them in a column family. In this approach, a new column family is created and multiple rowkeys are grouped in the column family. For example, *tag* family of table Post contains all the tag ids of the tags that have tagged the post. Then getting all the tags of a post can be divided into steps: get the tag ids by the rowkey-pid and fetches all these tags by rowkey.

Many- many relationship

In relational database, this kind of relationship is maintained by taking a new table that has foreign keys of both the tables. The same approach of one-to-many relationship is used in this case as well. Now we create one column family on each side of the relationship and add all the foreign keys in it. The third table which was earlier used to store foreign keys are removed from the system. Two new column families are added in User table that will contain rowkeys of the Post table and Comment table. Two column families are added in Post table that will contain rowkeys of comment and tag table. Figure 5.7 represents target HBase schema after applying transformation rule 3.

```
User:
Rowkey[uid]{
      CF[info]{
            Column[realname]:
                  value
            Column[emailid]:
                  value
            Column[homeurl]:
                  value
            }
      CF[credentials]{
            Column[username]:
                  value
            Column[pswd]:
                  value
            }
      CF[posts]{
            Column:
                  value
                  .
                  .
            }
      CF[comment]{
            Column:
                  value
                  .
                  .
            } }
Post:
Rowkey[pid]{
      CF[info]{
            Column[uid]:
                  value
            Column[title]:
                  value
            Column[time]:
                  value
            }
      CF[text]{
            Column[body]:
                  value
            Column[primary_url]:
                  value
            }
      CF[comment]{
            Column:
                  value
                  .
                  .
            }
      CF[tag]{
            Column:
                  value
                  .
                  .
            } }
Comment:
Rowkey[cid]{
      CF [info]{
            Column [postid]:
                  value
            Column [parentid]:
                  value
            Column [uid]:
                  value
            }
      CF[text]{
            Column [title]:
                  value
            Column[text]:
                  value
            }
      CF[score]{
            Column [descriptor]:
                  value
            Column [score]:
                  value
            } }
Tag:
Rowkey[tname]{
      CF [posts]{
            Column:
                  value
                  .
                  .
            } }
```

Figure 5.6: Target Schema transformation after rule 3

4. Merge interdependent tables

There are two types of table in relational databases. Some tables are more important and independently used than other tables which are only accessory and depend on other tables. Former tables are called main tables. The tables that are dependent on these tables and do not have any independent identification are called attached tables. These tables are merged into main tables by replacing a foreign key column or column family with the complete table in main table. For relational database, it is guaranteed that data is always in a consistent state and user data can not violate any constraint. For example, a comment cannot be uploaded with a user name which does not exist yet. But in HBase, system has to maintain these references instead. When a comment is deleted from the system, first delete it from table *Comment* and then from the *comment* column family of table *User*. On the other side, if an object does not refer other objects in the system, we do not have to link these two object, e.g. if the post system does not have the requirement that list all the comments of a given user, we can omit the column family *comment* which contain s the rowkeys.

5.2.2 Schema mapping

Schema Mappings are specifications that model a relationship between two data schemas. It transforms a source database instance into an instance that obeys a target schema. It is typically formalized as a triple (S, T, Σ) where S is a source schema, T is a target schema, and Σ is a set of dependencies and constraints that specify the relationship between the source schema and the target schema. A schema is a collection of named objects which provides a logical classification of objects in the database. Some of the objects that a schema may contain include tables, views, aliases, indexes, triggers, and structured types.

Sufficient work in this area has been done for both relational and XML databases but not in the case of column-oriented databases. In PostgreSQL, data is organised in tables and tuples whereas in HBase data is organized in tables, column family and columns. PostgreSQL stores data in row fashion and HBase in columnar fashion. Structure of HBase is somehow similar to that of XML, which is nested. So an approach based on nested mapping has been used for HBase.

5.2.2.1 Element Correspondence

After the schema transformation phase, we have both source and target schemas and it is important to relate these schemas now. This is known as Schema matching. An implementation of a schema matching process may use multiple match algorithms or matchers. This allows the selection of the matchers depending on the application domain and schema types. Given that the use of multiple matchers may be required there are two sub problems. First, there is the design of individual matchers, each of which computes a mapping based on a single matching criterion. Second, there is the combination of individual matchers, either by using multiple matching criteria (e.g., name and type equality) within a hybrid matcher or by combining multiple match results produced by different match algorithms [47]. For individual matchers, the following criteria are considered for classification:

- *Instance vs. schema:* Matching approaches can consider instance data (i.e., data content) or only schema-level information.
- *Element vs. structure matching:* Match can be performed for individual schema elements, such as attributes, or for combinations of elements, such as complex schema structures.
- *Language vs. constraint:* A matcher can use a linguistic approach (e.g., based on names and textual descriptions of schema elements) or a constraint-based approach (e.g., based on keys and relationships).
- *Matching cardinality*: Each element of the resulting mapping may match one or more elements of one schema to one or more elements of the other, yielding four cases: 1:1, 1: n, n:1, n:m. In addition, there may be different match cardinalities at the instance level.

Element matching technique is used to extract the element correspondence. In this, element of source schema is matched with element of target schema. By putting source schema is on one side and target schema is on another side, elements are matched with one another using arrows. The following figure shows the correspondence between source and target schema.

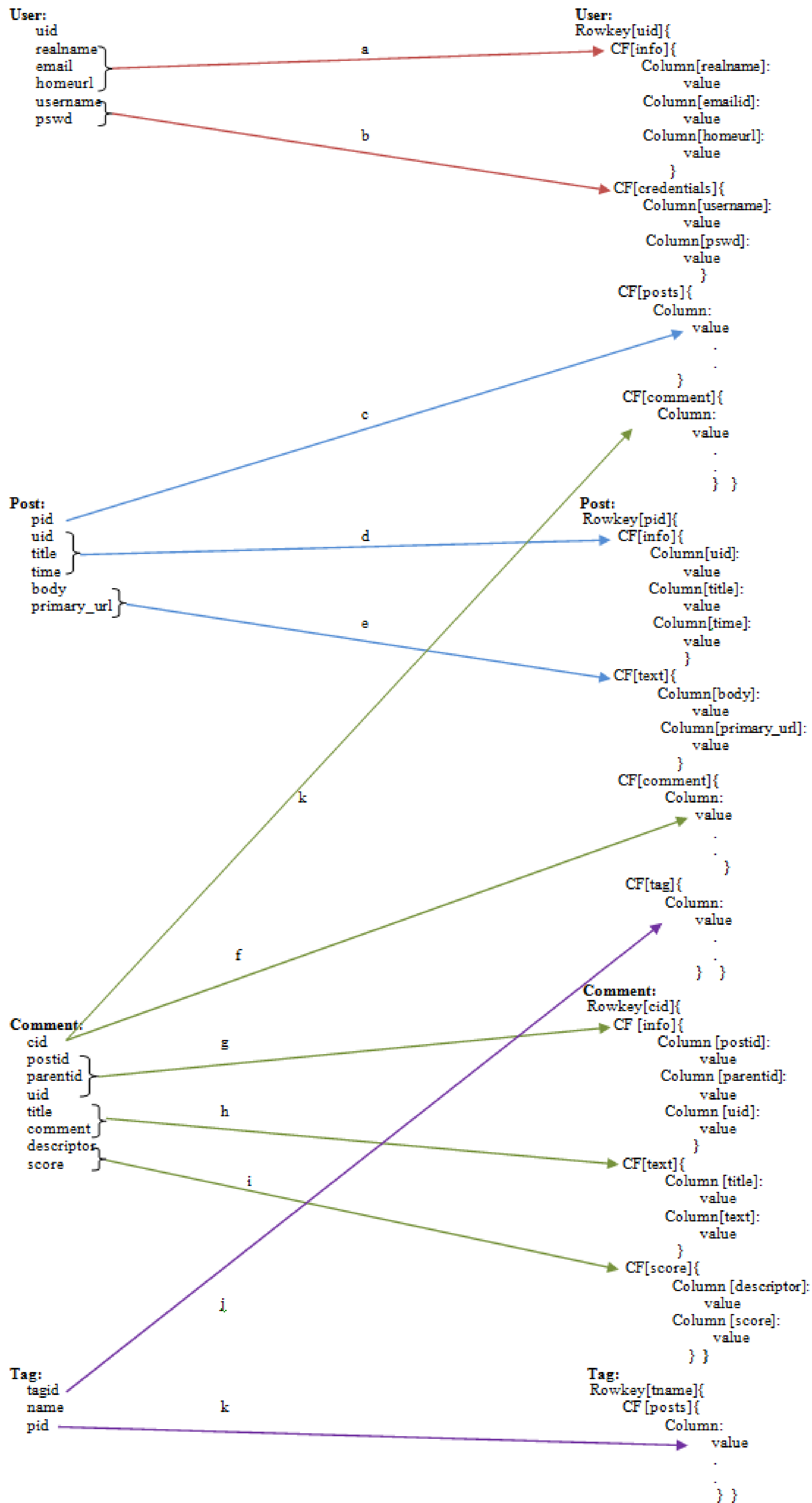


Figure 5.7: Schema correspondences between source schema and target schema

5.2.2.2 Tableaux

A tableau is a set of schema elements or attributes that are semantically related. Elements are repeated, for instance, when they are siblings under the same repeating element, or when the containing repeating elements are related via a parent-child relationship. Intra-schema constraints (e.g., key/foreign keys) are also used to define tableaux by chasing existing tableaux over the constraints. There is a set of tableau for both source schema and target schema. Each tableau contains all concepts that must exist together according to the referential constraints.

Tableau is generated for both the source and target schema after elements are matched. Tableau of source schema is generated the same way as for relational schema. But there is a different way of computing tableaux for HBase target schema. There are two types of column in HBase and these are treated differently in schema. First type is similar to relational columns and contains meta data of table. Only column value can have user data. For example, user id, realname,username, all are of first type. They contain fixed data. While in second type both column key and value can have user data. In the user table, *post* family contains column key and column value which contains post id of the post posted by the user. Thus this type of columns must be nested when generate the mappings. Tableaux for source and target schema are represented in below.

Computation of tableaux

Given the two schemas, the sets of tableaux are generated as follows. For each relation T in a schema, *primary path* are created that spells out the navigation path from the root to elements of T. For each intermediate set, there is a variable to denote elements of the intermediate set.

X1, X2, X3 and X4 are primary paths corresponding to the four tables associated with User, Post, Comment and Tag in the source schema. Note that in X3, the parent table User and Post are also included, since it is needed in order to refer to an instance of Comment and likewise in X4, the parent table post is included to refer to an instance of tag. Similarly, B1, B2, and B4 are primary paths in the target.

Y1, Y2, Y3 and Y5 are primary paths in target schema. In addition to the structural constraints (parent-child) that are part of the primary paths, the computation of tableaux also takes into account the integrity constraints that may exist in schemas.

These constraints are explicitly enforced in the tableaux Y4 and Y6. The tableau is constructed by enhancing, via *chase.*

For each schema, the set of its tableaux is obtained by replacing each primary path with the result of its chase (with all the applicable integrity constraints. For each tableau, for mapping purposes, we will consider all the atomic type elements that can be referred to from the variables in the tableau. Such elements are covered by the tableau. Thus, a tableau consists of a sequence of generators and a conjunction of conditions.

Tableaux for source schema:

X1= {u[uid, username, realname, email, homeurl, pswd] IN user;}

X1 represents that the users can exist in the system without post, comment or tag. It is not compulsory for a user to comment or post. Therefore, if we want to fetch a user information, we can directly infer a query on user table.

X2= { p[pid, uid, time, title, body, primaryurl] IN post; }

X2 represents the same logic as X1 that post can be there without any user, tag or comment.

X3= { u[uid, username, realname, email, homeurl, pswd] IN user ,p[pid, uid, time, title, body, primaryurl] IN post,c[cid, parentid, postid, uid, title, comment, score, descriptor] IN comment; p.pid=c.postid, u.uid=c.uid;}

Comment can only exist if an authorized user has written it and it is under a post or comment. Therefore, to fetch a comment requires checking of its parent post or comment and an authorized user.

X4= { p[pid, uid, time, title, body, primaryurl] IN post, t[tagid, name, pid] IN tag ; t.pid=p.pid;}

A tag can be there in the system only if it has posts tagged in. Therefore, to fetch a tag information, we have to check the posts tagged under this tag.

Tableaux for target schema:

Y1= { u[rowkey, username, realname, email, homeurl, pswd] IN user;}

Y2= { p[rowkey, uid, time, title, body, primaryurl] IN post; }

Y1 and Y2 represent the same logic as X1 and X2.

Y3= { u[rowkey, username, realname, email, homeurl, pswd] IN user, p[rowkey, uid, time, title, body, primaryurl] IN post, c[rowkey, parentid, postid, uid, title, comment, score, descriptor] IN comment; c.uid=u.rowkey, c.postid=p.rowkey;}

Comment can only exist if an authorized user has written it and it is under a post or comment. Therefore, if we want to fetch a comment, we have to check if it is under a post or comment and it is posted by an authorized user.

Y4= { u[rowkey, username, realname, email, homeurl, pswd] IN user, c2 [column,value] IN u.comment, p[rowkey, uid, time, title, body, primaryurl] IN post, c1 [column,value] IN p.comment, c[rowkey, parentid, postid, uid, title, comment, score, descriptor] IN comment; c1.column=c.rowkey, c2.column=c.rowkey, c.uid=u.rowkey, c.postid=p.rowkey;}

In Target schema, user also contains information of comments and posts posted by it. Therefore, bidirectional fetch is also possible.

Y5= { p[rowkey, uid, time, title, body, primaryurl] IN post, t[rowkey] IN tag, p1[column,value] IN t.post; p.rowkey=p1.column;}

A tag can be there in the system only if it has posts tagged in. Therefore, to fetch a tag information, we have to check the posts tagged under this tag.

Y6= { p[pid, uid, time, title, body, primaryurl] IN post,t1 [column,value] IN p.tag, t[rowkey] IN tag, p1[column,value] in t.post; p.rowkey=p1.column, t1.column=t.rowkey;}

To fetch a tag information, a post should be under it and the post should also contains the tag information. In target schema, post table also contains information about tags.

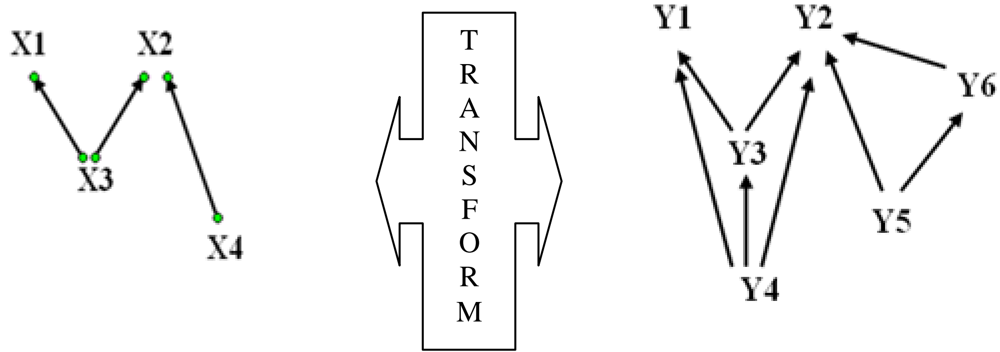


Figure 5.8: Tableaux hierarchy of source and target tableaux

5.2.2.3 Element Mapping

Triple (*X, Y, S*) defines a basic mapping: for all the data covered by *X*, there exist data covered by *Y* along with conditions that encode the correspondences. For each pair of tableau (*X, Y*), where *X* is a source tableau and *Y* is a target tableau with *S* be the set of all correspondences for which the source element is covered by *X* and the target element is covered by *Y*. Mapping (X, Y, S) is also written as ∀X → ∃ Y.S

Nested Mapping

The main concept is sub tableau in which tableau X is a sub tableau of a tableau X' if the data in X form a superset of the data covered by X' and also the conditions in X are a superset of those in X'. Then a basic mapping is ∀X1 → ∃ Y1.S nested inside another basic mapping ∀X2 → ∃ Y2.S if X2 and Y2 are strict sub tableaux of X1 and Y1 respectively, S2 is a strict superset of S1 and there is no correspondence S in S2 and V1 whose target element is covered by X1.
After applying all the above rules, we derived 4 nested mappings and query-like notations that represent the nested mappings, which can be used to transform the data.

M1: ∀X1→ ∃ Y1.[({a,b,d,e,g,h,i}∧∀(X3-X1) →∃ (Y3-Y1).{c,d,e,f,g,h,i,j,k}]

In target database, the user table contains information about user and information of comments and posts posted by the users. The user information is extracted from user table of source schema but comment and post information is fetched only after we apply a query to fetch all the posts and comments written by the user. Tableaux is used to represent the above query and element correspondence to match the elements of source to elements of target.

M1: user =for u IN User

```
        return [
            rowkey = u.uid
            info:realname = u.realname
            info:email = u.email
            info:homeurl = u.homeurl
            credentials:username  =  u.username
            credentials:pswd = u.pswd
            post = for p IN Post
                    where u.uid = p.uid
                    return [
                        value = p. uid ]
            comment = for c IN Comment
                        where u.uid = c.uid
                         return[
                                value = c.uid ] ]
```

M2: $\forall X2 \rightarrow \exists Y2.[(\{a,b,d,e,g,h,i\} \wedge \forall (X3\text{-}X2) \rightarrow \exists (Y3\text{-}Y2).\{a,b,c,f,g,h,i,k\}]$

The post table contains information about post and information of user, comments and tags related to a post. The post information is extracted directly from the post table of source schema but information of user, comments and tags is fetched by apply a query to fetch all the tags and comments. Tableaux is used to represent the above query and element correspondence to match the elements of source to elements of target.

M2: post = for p in Post

```
        return [
            rowkey = p.pid
            info:uid = p. uid
            info:title = p.title
            info:time = p.time
            text: body = p.body
            text:primary_url = p.primary_url
            comment = for c IN Comment
                        where p.pid = c.postid
```

```
                        return[
                                  value = c.cid ]
        tag = for t IN Tag
                 where p.pid = t.pid
                 return [
                            value = name ] ]
```

M3: ∀ X3→ ∃ Y3.[({a,b,d,e,g,h,i}∧∀((X3-X1)-X2) →∃ ((Y3-Y1)-Y2).{c,k,f,g,h,i,j}]

The comment table contains information about comments only. But a comment can only exist if it posted by a user and commented under a post or comment. Therefore, the comment information is extracted directly from the comment table of source schema but to check the user and the post, a query is need to be inferred that is represented in tableaux form.

M3: comment = for c IN Comment

```
                  return [
                      rowkey =c. cid
                      info:parentid = c.parentid
                      info:postid = c.postid
                      info:uid = c.uid
                      text:title = c.title
                      text:comment = c.comment
                      score:score = c.score
                      score:descriptor = c.descriptor ]
```

M4: ∀ X4→ ∃ Y4.[({d,e,l}∧∀(X4-X2) →∃ (Y4-Y2).{f,j,l}]

The tag table contains information about tag and information of posts tagged under a tag. The tag information is extracted directly from the tag table of source schema but information of posts is fetched by apply a query that fetch all the posts.

M4: tag = for t IN Tag

```
          return [
              rowkey = t.tagid
              post = for p IN Post
              where t.pid = p.pid
```

```
return [
    column = p.pid
    value = p.title ] ]
```

5.3 Target HBase Database in the form of tables

The output of the schema transformation and mapping of PostgreSQL database is HBase database and it is represented in the form of tables.

POST TABLE:

Row Key	Info			Text		Tags			Comment				
	Uid	Title	Time	Body	Primary_Url	t1	t2	t3	c1	c2	c3	c4	c5
p1	101	space shuttle endeavour's final journey	13-oct-2012 04:39 am	after over 296 days in space, nearly 123 million miles travelled	http://latimesblog.html	t1			c1	c2			
p2	102	making driverless cars safer	13-oct-2012 03:20 am	several autonomous cars have been developed elsewhere	http://people.csal.mit.edu.pdf		t2				c3	c4	
p3	102	apple and google	23-oct-2012 11:55 pm	some post related with apple and google”			t2	t3				c4	c5

Table 5.2: Post table

TAG TABLE:

Rowkey	Post		
	p1	p2	p3
Space	p1		
Google		p2	p3
Apple			p3

Table 5.3: Tag table

COMMENT TABLE:

Row Key	Info			Text		Score	
	Parentid	Postid	Userid	Title	Comment	Score	Descriptor
c1		p1	102	"Seems good"	"comment on 123 million miles travelled "	2	Informative
c2	c1	p1	101	"Re: Seems good"	"In reply to comment c1"	1	
c3		p3	101	"Wow apple and google together"	"comment on apple and google "	3	Funny
c4	c3	p3	102	"Why not"	"In reply to comment c3"	2	Decriptive
c5	c4	p3	101	"Of course"	"In reply to comment c4"	4	Insightful

Table 5.4: Comment table

USER TABLE:

Row Key	Info			Credentials		Post			Comment				
	Realname	Email_id	Homeurl	Username	Pswd	p1	p2	p3	c1	c2	c3	c4	c5
101	a0real	a0@ex.com	a0.ex.com	a0	a0p	p1				c2	c3		c5
102	a1real	a1@ex.com	a1.ex.com	a1	a1p		p2	p3	c1			c4	

Table 5.5: User table

Chapter 6: Conclusion and Future Scope

6.1 Conclusion

A transformation algorithm has proposed which takes input as relational database and transforms it into a column-oriented database. PostgreSQL have been used for relational database and HBase have been used for column-oriented database for implementing the proposed algorithm and the use case. The presented work also focuses on column-oriented databases and their representations. Transformation is divided into two steps, first step transforms the relational schema into column-oriented schema and the elements of relational database are matched with the elements of column-oriented database in the next step. Tableau is used for representing both source and target schema. Using these tableaux, nested mappings are generated that transform data from PostgreSQL database to HBase.

Conclusions drawn from the work done in this thesis can be summarized as:

- NoSQL databases have many advantages over relational database and more and more companies are opting for NoSQL databases.
- Among all the type of NoSQL databases, column-oriented databases are appropriate for OLAP like workload. Column-oriented databases perform better than row-oriented databases in Web applications.
- After comparison of HBase and Cassandra, it has been concluded that HBase has broader scope than Cassandra.
- The proposed transformation algorithm performs appropriately for a use case in which PostgreSQL database is considered as source database and HBase database is considered as target database.

6.2 Future Scope

Possible research problems that can be carried out further, can be:

- In this thesis, queries that was run on the source schema was chosen manually. This process can be automated for better performance. By just entering source database and matching the elements, target database can be generated.

- Algorithm can be extended to work on other type of column-oriented databases also, like Cassandra.
- On similar lines, algorithms can be proposed for transforming relational databases to other classes of NoSQL databases like document-oriented and graph-based databases.

References

[1] E.F.Codd, “A relational model of data for large shared data banks," Commutication of the ACM, 1970.

[2] The neo database, 2006, Nov. [Online]. Available: http://dist.neo4j.org/ Neo-technology-introduction.pdf [Accessed: Aug. 2012].

[3] S.Mazzocchi, Stefano's linotype, 2004, Feb. [Online]. Available: http:// www.betaversion.org/_stefano/linotype/news/46/ [Accessed: Sep. 2012].

[4] C.Strozzi, “Nosql a relational database management system”, [Online]. Available:http://www.strozzi.it/cgi-bin/CSA/tw7/I/en-US/nosql/Home%20Page [Accessed: Sep. 2012].

[5] F.Changetal, “Bigtable: a distributed storage system for structured data," in Proc. of the 7th symposium on Operating systems design and implementation OSDI '06, Berkeley, CA, 2006, 205-218.

[6] A.Lakshman and P.Malik, “Cassandra a decentralized structured storage system," Technical Report, Cornell University, 2009.

[7] Kai Fan, "Survey on NoSQL", Programmer, 2010, 76-78.

[8] S. Gilbert and N. Lynch, “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," in ACM SIGACT News, vol. 33, 2002.

[9] R.Rees, ”Comparison of NoSQL databases”, [Online]. Available:

http: //www.thoughtworks.com/articles/nosql-comparison [Accessed: Nov. 2012].

[10] Oracle. Available: http://www.oracle.com/in/index.html

[11] MySQL. Available: http://www.mysql.com/

[12] OrientDB. Available:http://www.orientdb.org/

[13] DyanmoDB. Available: http://aws.amazon.com/dynamodb/

[14] SimleDB. Available: http://aws.amazon.com/simpledb/

[15] Redis. Available: http://redis.io/

[16] Cassandra. Available: http://cassandra.apache.org/

[17] Titan. Available: http://thinkaurelius.github.io/titan/

[18] MongoDB. Available: http://www.mongodb.org/

[19] CouchDB. Available: http://couchdb.apache.org/

[20] Guillem Rull, Carles Farre, Ernest Teniente, and Toni Urpi, “Validation of Schema Mappings with Nested Queries”, in ComSIS Vol. 10, No. 1, January 2013.

[21] Swati joshi, “Analysis Of Schema Matching Technique Using Xml Data For Performance Enhancement”, in International Journal of Computer Architecture and Mobility, Volume 1-Issue 4, February 2013.

[22] Philip A. Bernstein and Laura M. Haas, “Information integration in the enterprise”, in Communications of the Association for Computing Machinery (CACM), pp.72–79, 2008.

[23] Balder ten Cate, Phokion G. Kolaitis, and Wang-Chiew Tan, “Schema Mappings and Data Examples”, in ACM, 13 March 2013.

[24] Bogdan Alexe, Balder ten Cate, Phokion G. Kolaitis, and Wang Chiew Tan, “Designing and refining schema mappings via data examples.”, in SIGMOD, pp.133–144, 2011.

[25] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth., “Clio Grows Up: From Research Prototype to Industrial Tool”, in ACM SIGMOD, pp. 805–810, 2005.

[26] Popa, L., Velegrakis, Y., Miller, R.J., Hern andez, M.A., Fagin, R., “Translating Web Data”, in Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 598–609, 2002.

[27] Ronald Fagin, Laura M. Haas, Mauricio Hern andez, Renee J. Miller, Lucian Popa1, and Yannis Velegrakis, “Clio: Schema Mapping Creation and Data Exchange”, in Springer-Verlag Berlin Heidelberg, pp.198–236, 2009.

[28] Alessandro Raffio, Daniele Braga, Stefano Ceri, Paolo Papotti, Mauricio A. Hernandez, “Clip: a Visual Language for Explicit Schema Mappings”.

[29] S. Melnik, P. A. Bernstein, A. Halevy, and E. Rahm, “Applying Model Management to Executable Mappings,” in SIGMOD, 2005, pp. 167–178.

[30] M. Lenzerini, “Data Integration: A Theoretical Perspective,” in PODS, pp. 233–246, 2002.

[31] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa, “Data Exchange: Semantics and Query Answering,” in ICDT, pp. 207–224, 2003.

[32] R. Fagin, P. Kolaitis, L. Popa, and W.C. Tan, “Composing Schema Mappings: Second-Order Dependencies to the Rescue,” in PODS, pp. 83–94, 2004.

[33] C. Yu and L. Popa, “Semantic Adaptation of Schema Mappings when Schemas Evolve,” in VLDB, pp. 1006–1017, 2005.

[34] Bogdan Alexe, Balder ten Cate, Phokion G. Kolaitis, WangChiew Tan, “EIRENE: Interactive Design and Refinement of Schema Mappings via Data Examples”, in Proceedings of the VLDB Endowment, Vol. 4, No. 12, 2013.

[35] Li Qian, Michael J. Cafarella, H. V. Jagadish, “Sample-Driven Schema Mapping”, in SIGMOD ’12, May 20–24, 2012.

[36] Leopoldo Bertossi, Camilla Schwind, “Database Repairs and Analytic Tableaux”, 2003.

[37] Gosta Grahne and Adrian Onet, “On Conditional Chase Termination”, 2012.

[38] Libkin, “Data exchange and incomplete information.”, In PODS, pp. 60–69, 2006.

[39] Hernich, A., and Schweikardt, “Solutions for data exchange settings with target dependencies.”, In PODS. pp. 113–122, 2007.

[40] HBase. Available: http://HBase.apache.org/

[41] Philippe Rigaux, “Understanding HBase and BigTable”, [Online]. Available: http://jimbojw.com/wiki/index.php?title=Understanding_HBase_and_BigTable [Accessed: Feb. 2013].

[42] Ian Thomas Varley, “No Relation: The Mixed Blessings of Non-Relational Databases”, 2009.

[43] Olivier Cure, Myriam Lamolle, and Chan Le Duc, “Ontology Based Data Integration Over Document and Column Family Oriented NOSQL Stores”, 2011.

[44] Nick Dimiduk and Amandeep Khurana, “HBase in Action”, Manning Publications Co., 2012.

[45] “Understanding HBase— The data model”, [Online]. Available: http:// internetmemory.org/en/index.php/synapse/understanding_the_HBase_data_mode l, [Accessed: Feb. 2013].

[46] Lars George, “HBase: The Definitive Guide”, O'Reilly Media, pp. 45-89, 2011.

[46] Anisur Rahman, “Tabular Representation of Schema Mappings: Semantics and Algorithms”, 2011.

[47] Dimitris Manakanatas, Dimitris Plexousakis, “A Tool for Semi-Automated Semantic Schema Mapping: Design and Implementation”, in ACM, 2005.

[48] Stonebraker et al., “C-Store: A column-oriented DBMS”, Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005.

[49] Copeland, George P. and Khoshafian, “A decomposition storage model”, in SIGMOD '85, 1985.

[50] Daniel Abadi and Samuel Madden, "Debunking another Myth: Column-Stores vs. Vertical Partitioning", The Database Column, 31 July 2008.

[51] Stavros Harizopoulos, Daniel Abadi and Peter Boncz, "Column-Oriented Database Systems", in VLDB 2009 Tutorial, 2009.

[52] Anil Gupta, “HBase vs Cassandra”, [Online]. Available: http://bigdatanoob. blogspot.in/2012/11/HBase-vs-cassandra.html, [Accessed: Jan. 2013].

[53] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, and Wilson C. Hsieh, “Bigtable: A Distributed Storage System for Structured Data” in Seventh Symposium on Operating System Design and Implementation, 2006.

[54] Avinash Lakshman and Prashant Malik, “Cassandra - A Decentralized Structured Storage System”, in ACM, 2009.

[55] Cassandra storage engine, 2012, [Online]. Available: https://kb.askmonty.org/en-cassandra-storage-engine/, [accessed on: march 2013].

[56] Gcottman, 2010, “Cassandra Column Families”, [Online]. Available: http://wiki.toadforcloud.com/index.php/Cassandra_Column_Families, [accessed on: feb. 2013].

[57] Robey Pointer , 2010, “Introducing FlockDB”, [Online]. Available: https://blog.twitter.com/2010/introducing-flockdb, accessed on: jan. 2013].

[58] J Shoberg, “Building Search Applications with Lucene and Nutch”, in Apress, 2006.

[59] Tom white, “Hadoop: The Definitive Guide”, O'Reilly Media, pp. 524, 2009.

[60] Apache Solr, Available: http://lucene.apache.org/solr/.

Appendix

Pseudo-code for transformation algorithm

Global Parameters Passed:

i. String table[] - An array to store the table names.
ii. Sting col[]-An array to store column names of a table.
iii. String PK-A variable for storing the primary key of a table.
iv. String CF[]-An array for storing column families of a table.
v. String EK[]-An array for storing exported keys of a table.
vi. String main_tables[] and attached_tables[]-Two arrays for storing names of main and attached tables.

Function Name: **Main()**

Called By: Operating System

Calling:

i. get_table_names()
ii. get_column_names()
iii. get_primary_key()
iv. create_table()
v. group_correlated_columns()
vi. get_exported_keys()
vii. check_uniqueness()
viii. add_column()
ix. add_column_family()
x. find_main_tables()
xi. find_attached_tables()

Parameters Passed: Source database

Return: Target database

Function Name: **get_table_names()**

Called By: Main function

Calling: Nothing

Parameters Passed: Source database

Purpose: To extract all the table names from source database.

Function Body:

i. Finds all the table names.
ii. Stores table names in an array.
iii. Return to main() function.

Return: An array of table names.

Function Name: **get_column_names()**

Called By: Main function

Calling: Nothing

Parameters Passed: Table name

Purpose: To extract all the column names of the table.

Function Body:

i. Finds all the column names of a table.
ii. Stores column names in an array.
iii. Return to main() function.

Return: An array of column names.

Function Name: **get_primary_key()**

Called By: Main function

Calling: Nothing

Parameters Passed: Table name

Purpose: To get primary key of the table.

Function Body:

i. Find primary key of the table by firing SQL queries on source database.
ii. Return to main() function.

Return: primary key

Function Name: **create_table()**

Called By: Main function

Calling: Nothing

Parameters Passed: Table name, array of column names and primary key.

Purpose: To create a new table in target database.

Function Body:

i. Create new table in the target database with table name same as input table name.
ii. Input array of column names is used to create new columns in the target table.
iii. Set rowkey of the table as primary key of the source the table.
iv. Return to main() function.

Return: Name of the new table created in the target database.

Function Name: **group_correlated_columns()**

Called By: Main function

Calling: Nothing

Parameters Passed: An array of column names.

Purpose: To create column families.

Function Body:

i. Group correlated columns and create column family for each such group.
ii. Put all the column families in an array.
iii. Return to main() function.

Return: An array of column families.

Function Name: **get_exported_keys()**

Called By: Main function

Calling: Nothing

Parameters Passed: Table name

Purpose: Find exported keys and their table names of input table.

Function Body:

i. Exported keys are found by firing SQL queries on source table.
ii. Place exported keys with their table names in a 2-d array.

iii. Return to main() function.

Return: A 2-d array containing exported keys and their table names.

Function Name: **check_uniqueness()**

Called By: Main function

Calling: Nothing

Parameters Passed: Table name and attribute name.

Purpose: check for duplicate values in a column.

Function Body:

i. If a column contains duplicate values it returns 1 otherwise 0.
ii. Return to main() function.

Return: A variable of int type.

Function Name: **add_column()**

Called By: Main function

Calling: Nothing

Parameters Passed: HBase table and rowkey.

Purpose: Create a new column.

Function Body:

i. Create a new column with column name as rowkey.

ii. Return to main() function.

Return: Nothing.

Function Name: **add_column_family ()**

Called By: Main function

Calling: Nothing

Parameters Passed: HBase table and rowkey.

Purpose: Create a new column family in the HBase table.

Function Body:

i. Create a new column family which will store multiple rowkeys.
ii. Return to main() function.

Return: Nothing

Function Name: **find_main_tables()**

Called By: Main function

Calling: Nothing

Parameters Passed: Source database

Purpose: find main tables in the target database.

Function Body:

i. Check for the main tables in the target database.
ii. Put the main tables in a separate array.
iii. Return to main() function.

Return: An array of main tables.

Function Name: **find_attached_tables()**

Called By: Main function

Calling: Nothing

Parameters Passed: Source Database

Purpose: Find attached tables in the target database.

Function Body:

i. Check for the attached tables in the target database.
ii. Put the attached tables in a separate array.
iii. Return to main() function.

Return: An array of attached tables.