Data Storage and Retrieval for Social Network Services

by

© Wang Tao

A thesis submitted for the degree of

Master of Science

School of Computing

National University of Singapore

2010

Abstract

In recent years, social network services have become ever more popular and even begin to affect people's life. A lot of social network sites have attracted tens of millions of users, where people contribute content, share information and activities with each other. Social network services are so popular as they allow users to display their creativity and knowledge, take ownership of the content, and obtain shared information from the community. A social network site serves as a platform for users of a community to interact and collaborate with each other. In social networks, users are connected through various social relationships like friendship, professional, academic and etc., while a huge amount of objects such as blogs, photos and videos are connected to the users through ownership, comment-relationship, tagging-relationship and so on. Obviously, a social network contains extremely complicated relationships. This brings many challenges for querying and analyzing social network data.

The popularity of social network services and the challenges for querying and analyzing social network data have driven to develop a new type of systems to support social network services. In this thesis, we focus on investigating a new data storage and indexes for a new graph database which is designed to manage nonblob data for social network services. We introduce two approaches, the Ordering method and the Minimum Spanning Tree(MST) method, to partition a huge social network graph into several small parts and distribute them over a cluster of servers. Two types of indexes, content index and node index, are investigated to improve the performance. We also design an object store system, called HadoopObS, to store blob data for social network services. Several experiments on crawled Flickr data are conducted to evaluate our storage and index design.

Acknowledgements

I am heartily thankful to my supervisor, Professor TAY Yong Chiang, for his encouragement, guidance and support for this work.

It is a pleasure to thank Dai Bingtian and Lin Yuting who configured and maintained Awan cluster for me to conduct the experiments. I would like to offer my regards and blessings to all of my friends who supported me in any respect during the completion of this work.

Wang Tao

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, social network services have become ever more popular and even begin to affect people's life. A lot of social network sites(SNSs) such as Fackbook[1], Flickr[2], Delicious[3] and MySpace[4] have attracted tens of millions of users, where people contribute content, share information and activities with each other. Social network services are so popular as they allow users to display their creativity and knowledge, take ownership of the content, and obtain shared information from the community. A social network site serves as a platform for users of a community to interact and collaborate with each other. In social networks, users are connected through various social relationships like friendship, professional, academic and so forth, while a hug amount of objects such as blogs, photos and videos are connected to the users through ownership, comment-relationship, tagging-relationship and so on. Obviously, a social network contains extremely complicated rela-

[1] http://www.facebook.com

[2] http://www.flcikr.com

[3] http://delicious.com/

[4] http://www.myspace.com/

tionships and this brings many challenges for querying and analyzing social network data.

1.1 Motivation

Data of social network services have several differences with conventional data which are usually stored as tables in relational databases. As we mentioned, social network data contain extremely complicated relationships, but traditional databases have troubles in representing complex relationships as they use the simple table structures to store data. However, in relational model, relationships are based on set theory and must be recovered by executing join operations on the database due to lacking explicit representation, while join operations are expensive. In 1977 Leinhardt first introduced the idea of using a directed graph to represent a social community[35]. A directed graph is a pair $G = (V, E)$ where V is a set of vertices or nodes while E is a set of ordered pairs of vertices called directed edges or simply edges. Figure 1.1 is a sample of an acyclic directed graph which represents a small social graph of Flickr[2]. A graph representing a social network has some basic structural properties and these properties are very useful for analyzing and querying a social network. Every day terabytes data are uploaded to Facebook and more than 25 terabytes of data are managed by Facebook. Traditional databases are designed for efficient transaction processing such as updating, inserting and retrieving small number of information in a large database, however, they will suffer serious problems when trying to retrieve or analyze a large amount of information[26].

Consequently, traditional databases incur troubles in managing and querying the data of social network services and these have generated challenges to the research community

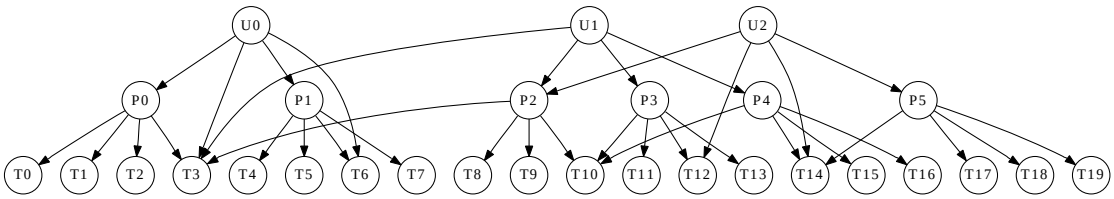

Figure 1.1: A Sample Acyclic Digraph. The nodes labeled by $U_i(i = 1, 2, 3)$ denote users, while the nodes labeled by $P_i(i = 1, 2, ...)$ or $T_i(i = 1, 2, ...)$ are photos or tags respectively. A directed edge (U_i, P_i) means user U_i uploaded photo P_i, (U_i, T_i) denotes user U_i published tag T_i and (P_i, T_i) denotes photo P_i is tagged by tag T_i .

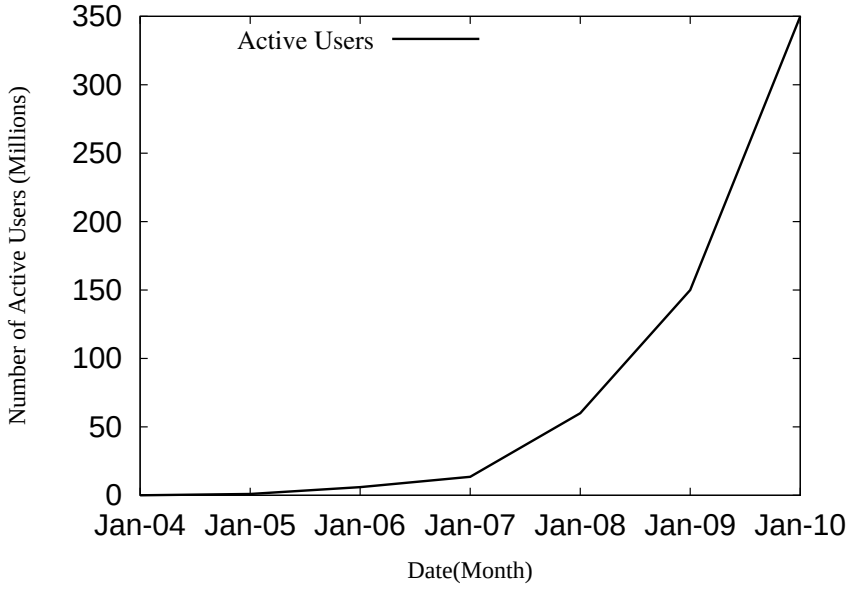

Figure 1.2: The Growth of Active Users on Facebook.

how to manage data in such scale. Besides, the number of users on SNSs is increasing rapidly and Figure 1.2 shows the growth of active users on Facebook is quite fast. Facebook has surpassed Google to be the most popular site in terms of total worldwide visitors to their Web sites as shown in Table 1.1 and there are three sites that are social network sites in the

Rank	**Domain**	**Visits**	**Unique Visitors**	**Page Views**
1	facebook.com	2,712	132	140,607
2	google.com	2,686	146	37,458
3	yahoo.com	2,556	133	56,590
4	live.com	1,253	76	16,626
5	msn.com	1,083	85	8,614
6	aol.com	698	56	17,025
7	ebay.com	657	88	13,989
8	youtube.com	559	91	8,265
9	myspace.com	554	49	43,162
10	amazon.com	418	85	7,135

Table 1.1: Top 10 Web Sites According to Compete[1](Millions)

top 10 sites. There are more than 2,712 million of visitors on Facebook every month and these visitors submit millions of queries every hour. This has brought large opportunities as well as challenges for research in social network services and driven the design of new data models and storage platforms which impose the requirements of social network services.

In addition, a major characteristic of social network services is folksonomy, which is also

known as collaborative tagging. Tag-based applications in social network services are becoming popular, and millions of users are using billions of tags to label public resources. Most queries currently supported by these applications are keyword-based, and the results returned by the system may not be precise and meaningful. In consequence, the new systems should provide more precise and meaningful results in an efficient way.

1.2 Objective

The popularity of social network services and the limitations of existing systems to support such services have driven to develop a new type of systems to support social network services. This leaves open the following research topics:

1. Data Model

   Investigate a new data model and corresponding operations for the data prevalent in social network services. The new data model should represent the new features of such data and support them better.

2. Storage Design

   Evaluate existing storage structures and design a new storage structure to support the new data model for social network services. Build a distributed data storage system with high availability and scalability based on the new storage structure. This storage system should implement efficient data manipulation, meta-data management, replication and failure recovery.

3. Indexing

Indexing is the most important and fastest approach which reduces high I/O cost effectively and greatly improves the speed of data retrieval operations. Therefore, it is important to design indexing mechanisms for the new storage structure.

4. Query Processing

   Social network services typically support millions of users, such as Facebook has more than 350 million active users, and these users may submit millions of queries per hour. To handle workload of this scale, an efficient query processor should be developed.

In these four topics, we focus on the storage design and indexing. In this thesis, the data storage problem is divided into two subproblems, nonblob data storage problem and blob data storage problem.

1.3 Contribution

This thesis makes the following contributions:

1. Data Model and Storage

   Investigate a novel graph data model and storage for nonblob data in social network services.

2. Data Partition

   Social network graphs are extremely large, therefore, it is important to partition them into small pieces and we will propose two partition methods, the Ordering partition method and the MST partition method.

3. Indexes

   Indexing is the most important and fastest approach which reduces high I/O cost effectively and greatly improves the speed of data retrieval. We introduce two types of indexes: content index and node index.

4. Blob Data Storage

   Beside the nonblob data storage problem, the blob data storage problem is also important for social network services. For instance, Facebook has more 80 billion image files which are hundreds of petabytes in total.

1.4 Organization

The rest of this thesis is organized as follows. We survey some current storage structures of existing database systems, such as relational databases, Bigtable, PNUTS, semi-structured model and so forth, and analyze the advantages and disadvantages for each storage structure and limitations in supporting social network services in Chapter 2. Chapter 3 introduces the architecture of our system which consists of a graph database system and an object store system. We propose the graph data model, data storage and indexes of our graph database system in Chapter 4, while the object store system which is designed to sore blob data is described in Chapter 5. In Chapter 6, we conduct some experiments to evaluate our storage and index design for both nonblob data and blob data. Finally, we makes a conclusion and a sketch of future work in Chapter 7.

Chapter 2

Related Work

2.1 Relational Database

Relational data model is the most popular data model and can be supported by serval types of storage systems, such as: Row Store, Column Store and so on. Relational databases have been the predominant database systems since the 1980s and achieved a great success.

Unfortunately, this conventional relational model still has some limitations and these limitations can be divided into three categories:

1. Fundamental Limitations

   The conventional relational model has several limitations which are the fundamental shortcomings of the relation model.

   (a) Lack of Object Identity

   In the relational databases, there is no independent identification of existence

for entities. The database systems identify and access objects indirectly via the identification of the attributes which characterize them. In practice, relational systems strive for supporting permanent and inspectable object identification techniques.

(b) Lack of Explicit Relationship

In the entity-relationship model, explicit entities and relationships are specified. However, in the relational model, relationships are based on set theory and must be recovered by executing relation operations on the database due to lacking explicit representation. As shown in Figure 2.1, a relationship(*Comment*) connects two entities(*User* and *Photo*) together, but in the relational model, there are only three tables and no explicit representation of this relationship.

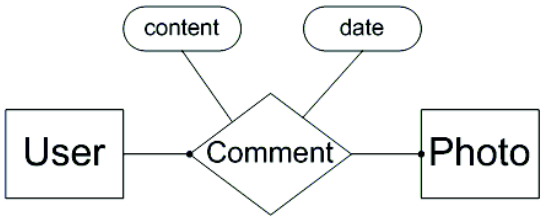

Figure 2.1: A Small E-R Diagram

2. Limitations in Special Forms of Data

Besides the fundamental limitations, there are many special forms of data which require special types of representation, such as temporal data, spatial data, unstructured data and so on.

3. Limited operations

Relational model has a fixed set of SQL operations, and this causes some computational problems, such as recursive queries are extremely difficult to be specified and implemented in relational databases.

RID	U_ID	U_NAME	U_AGE
1	1237	Jane	23
2	4322	John	34
3	1456	Jack	28
4	8765	Tom	40

Figure 2.2: A Small Sample Table

2.1.1 Row Store

Most major relational DBMSs are implemented on record-oriented storage system. Each record consists several attributes and these attributes are stored continuously on disk as Figure 2.3 shows. Obviously, high performance writes are achieved and DBMSs with row store architecture are called write-optimized system [41].

However, the row-store systems suffer problems in managing sparse tables which has been investigated a lot by research community in [12, 36, 31, 6]. This type of data is very popular in community system. For instance, Google Base has more than 400 million tuples which are defined by more than 3000 attributes while only less than 20 attribute are defined for each tuple. The massive presence of NULLs incurs massive redundant storage and causes performance problems in row store systems. Therefore, row-oriented relation databases incur serious troubles in managing this type of data due to the presence of a massive number

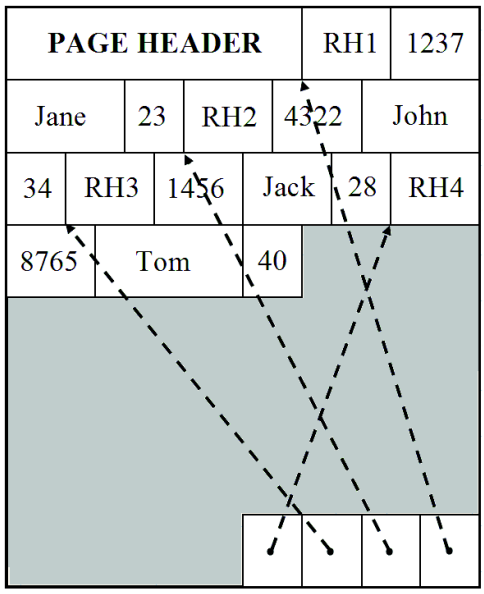

Figure 2.3: The Standard Page Format for Row-Store

of NULLs.

2.1.2 Column Store

Recently, several column-oriented database systems are implemented, including MonetDB[9] and C-Store[41]. Column-store systems store each column of a relation separately on disk, as shown in Figure 2.4 and use join indexes to reconstruct the original table. In C-Store, each relation is divided into several C-Store projections and each projection contains one or more attributes of the original table. C-Store also introduces some techniques to reduce disk storage cost and I/O cost, including sorting and compression. The major differences between row-store and column-store systems are typically concerned with the efficiency of hard-disk access for a given workload. Column-store systems are more efficient when operations are only on small number of attributes but a large number of rows.

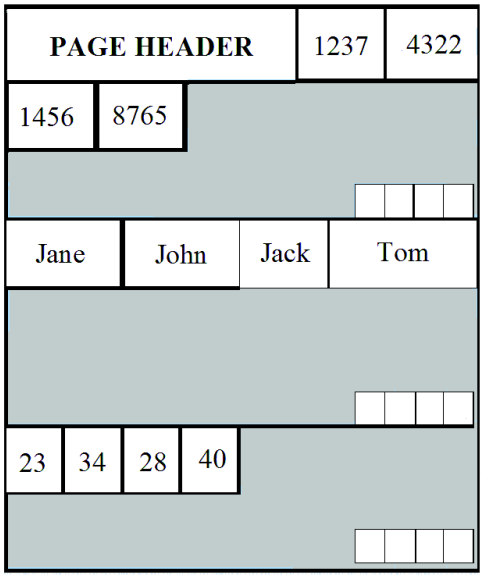

Figure 2.4: A Page Format for Column-Store. The responding table is shown in Figure 2.2.

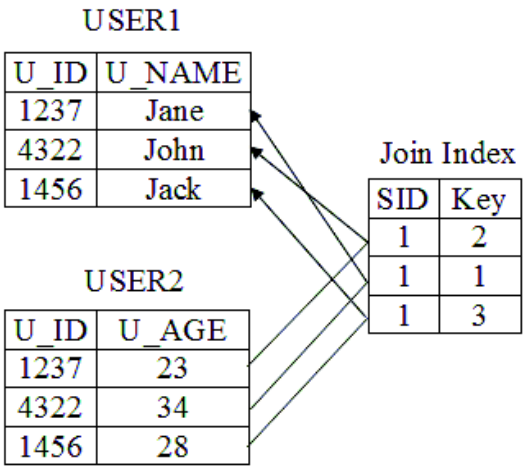

Figure 2.5: A Join Index Sample

However, column-store systems still have some limitations. In [24], some experiments are conducted and the results show that when the number of rows is held constant and the number of columns increases by a factor of eight, the scan time has not even doubled in standard row store but has increased by a factor of ten in column store. This is due to column-store systems have to reconstruct each rows when scan a table and this costs significantly even using join indexes. Besides these, column-store systems are still relational systems, hence, they still induce the limitations that relational model has.

2.2 Bigtable

Bigtable is a distributed storage system for managing structured data and was proposed in [12]. It is developed since 2004 and is now used by a number of Google projects, such as Google Maps[1], Google Book Search[2], Google Earth[3], Google Base[4] and YouTube[5]. A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map[12]. Each table consists of rows and columns, and each cell has a timestamp. Bigtable is designed to scale a very large size of data, and in order to manage huge tables, tables are horizontally partitioned into row ranges and each row range is called a tablet, which is the unit of distribution and load balancing. Bigtable is built on Google File System(GFS)[20], which is used to store data files. GFS is a distributed file system which has high performance, scalability, reliability, and availability.

[1] http://maps.google.com/

[2] http://books.google.com/

[3] http://earth.google.com/

[4] http://base.google.com/base/

[5] http://www.youtube.com/

Both the row store and column store which we have discussed are designed for low to medium dimensional dense datasets and have trouble managing high-dimensional data, while Bigtable handle this type of data well. For example, Google Base has more than 400 million tuples which are defined by more than 3000 attributes while only less than 20 attribute are defined for each tuple. The massive presence of NULLs incurs redundant storage and introduces another dimension of optimization. HBase [5] is an open-source, distributed, column-oriented store modeled after Google' Bigtable by Chang et. al. in [12].

However, Bigtable does not meet the normal requirements of an ACID [23] database for transaction processing with its limited atomicity, application-dependent consistency, uncertain isolation and excellent durability. Besides these, Bigtable is based on relational model, therefore, it still has some limitations that traditional relational model incurs, such as lack of object identity and explicit relationship. Consequently, Bigtable is also not suitable for managing data of social network services which contain a large number of objects and complicated relationships.

2.3 PNUTS

PNUTS is a massive-scale, hosted database system which aims to support Yahoo!'s web applications[17]. In PNUTS, data is organized into tables of records with attributes and presented to users as in relational databases. These data tables are horizontally partitioned into groups of records called tablets which is similar to Bigtable[12]. PNUTS stores tablets as storage units and storage units respond to a simple API of get, set and scan requests. Each storage unit manages a tablet that contains an interval either of the ordered table

key space or the hash table value space. The mapping from intervals to storage units is held permanently by the tablet controller which acts as a master for a PNUTS instance. These tablets are distributed across many nodes and each tablet contains thousands or tens of thousands of records. Each record has a primary key and an assigned owner, used to deliver PNUTSs consistency guarantees. A table's primary keys may be ordered or hashed, with ordering more naturally supporting range queries and hashing lending itself to load balancing. However, PNUTS is designed for online serving workloads in which most of the queries read and write single records or a small number of records.

The similarities and differences between PNUTS and Bigtable are as following:

- Similarities:

    1. Both PNUTS and Bigtable are based on relational tables with flexible schema.
    2. Some concepts in them are similar, such as record, tablet.
    3. Bigtable maintains data in lexicographic order by row key and records in PNUTS are ordered or hashed.
    4. Both PNUTS and Bigtable horizontally partition tables into tablets.

- Differences:

    1. Bigtable stores multiple versions of data using timestamps, while PNUTS does not.
    2. PNUTS supports indexes, such as hash index, but Bigtable has no indexes.

Obviously, PNUTS and Bigtable are very similar, although some differences exist. Both PNUTS and Bigtable are based on relational tables with flexible schema, hence, PNUTS

also has some limitations of traditional relational model and induces trouble in managing data of social network services as Bigtable. In addition, PNUTS and Bigtable induce troubles in managing data with complex relationships due to lacking explicit representation of relationships.

2.4 Semistructured Data Model and Storage

In semistructured model, there is no separation between the data and the schema. Semistructured model can well model the data sources which cannot be constrained by a schema such as Web and is extremely flexible for data exchange between disparate databases. Semistructured data is naturally modeled as graphs with labels which give semantics to its underlying structure.

Definition 2.4.1 *An edge labeled directed graph is a triple $G = (V, E, \ell)$ where V is a set of vertices, $E \subseteq V \times V$ is set of edges and $\ell : E \rightarrow L$ is a mapping from edges to a set of strings ℓ called labels.*

Object Exchange Model(OEM) and Extensible Markup Language(XML) are usually considered as standards of data representation and exchange on the World-Wide Web[22].

2.4.1 Object Exchange Model

Object Exchange Model(OEM) is first proposed in [37] and a basic data model which is used in several projects of the Stanford university Group, including Lore and C^3[21]. It is a model for exchanging semi-structured data between object-oriented databases and

designed for three goals: Information exchange, Information discovery and browsing, and Mediators[21].

2.4.2 Extensible Markup Language

Extensible Markup Language(XML) is a textual language which was developed for data representation and exchange on the Web[10]. Several approaches are investigated to query XML data such as XQuery[11], XPath[16] and etc. However, it is more challenging than storing XML data in relational databases. Because there are some fundamental mismatches between the XML structured data and the relational data model which major commercial RDBMS products support. A lot of work has been done by research community on storing XML data and these methods are usually divided into three categories:

1. Storing in Relational Databases

   Relational databases are the prevailing database system in commercial database market. It is very necessary and important to investigate storing XML data in relational databases. In relational databases, XML documents are parsed into tables or just stored as Binary Large Objects(BLOB). That is, there are two methods to store XML documents in relational databases.

   (a) Converting XML documents into tables

   XML documents are parsed and mapped into relational tables and XML queries are translated to SQL queries over these tables [19, 7, 40, 39, 42]. Each XML document can be represented as a labeled directed graph and each element in this XML document is a node. Subsequently, nodes and edges are converted

into tables. The major advantage of this method is that it is not required to modify existing database engines too much.

(b) Storing XML documents as BLOB

In this method, XML documents are stored as Binary Large Objects(BLOB) in columns of relational tables. This method is very simple and most commercial databases support it, such as Microsoft SQL Server, Oracle 10 and etc.. However, the major problem is that it is impossible to query the details of XML documents and any operation on these XML documents has to load the entire XML document to main memory first.

2. Storing in Native XML Data Management Systems

In native XML data management systems, XML documents are stored according to XML data model in a tree structure and only XQuery is supported.

3. Storing in XML-Relational systems

This is a hybrid method. XML documents are stored on logical pages in tree structures matching the XML data model[25, 8]. It does not need to map XML documents into relational tables but encode XML documents into relational tables.

In native XML data management systems, many XML index algorithms are proposed and can be classified into four categories: node indexes [13], content indexes[32], path indexes [18, 15] and hybrid indexes [44, 28]. Node indexes are used to efficiently support Structural Join (SJ) and Holistic Twig Join (HTJ). Path indexes use structural summaries to provide efficient accesses to nodes which satisfy certain structural relationships like $parent/child$. In contrast, content indexes provide efficient accesses to the text or the attribute values of

nodes and these content indexes can be implemented using B-trees or inverted lists. Hybrid indexes are a hybrid approach for indexing both structure and content at a time and also called content-and-structure (CAS) indexes.

However, semistructured model is designed for data exchanging between disparate databases and on the World-Wide Web. Therefore, it has some limitations in storing and querying social network data. The hierarchical structure is suitable for most documents but not suitable to represent non-hierarchical relationships, such as many-to-many relationships. In consequence, it is a limited representation of relationships. In addition, XML does not support explicit representation of intrinsic data types such as integer, string, boolean and so on. It is more difficult to query information in semistructured model due to XML documents need to be parsed first.

2.5 Object-Oriented Database

Object-oriented concept was first introduced in programming languages. The discovery of the limitations of the relational databases and the need of managing a large number of objects in object-oriented programming languages led to introduce object-oriented concept to database systems, that is, object-oriented database systems[29]. Therefore, object-oriented databases(OODB) add database functionality to object programming languages. OODBs extend the semantics of the C++, Smalltalk and Java object-oriented programming languages to provide full-featured database programming capability, while retaining native language compatibility. In OODBs, a database is considered as a collection of objects whose behavior, state, and relationships are stored as a physical entity[45]. Compared with

RDBs, OODBs have several advantages:

1. OODBs are more realistic and powerful, especially in handling complex objects. Entities in real world are more naturally modeled as objects than tables. OODBs can handle a large collection of complex data due to user can define and add new data types based on the predefined data types.

2. In OODBs, relationships can be inherited among sets of entities.

3. OODBs are fast in querying complex data structures and use expressive queries for accessing data.

4. OODBs have more powerful data operations. OODBs are computationally complete by binding to existing object-oriented programming languages and these data operations are not limited several SQL operations[33].

OODBs can be divided into two categories: stand-alone OODB, and OODB with existing Data Sources according to different application environment. A stand-alone OODB system is a system where OODB model is used in both the database and the application therefore, no data mapping is needed between the database and the applications. However, in a OODB system with existing data sources, data mapping is needed. The non-object data is mapped into object models and stored in the OODB.

The correspondence of the basic terms in relational and object-oriented databases is shown in Table 2.1. The first three terms are similar between relational and object-oriented databases although there are still some differences between them. However, a method is very different with a stored procedure for the fourth term of two types of databases. Methods are

Object-oriented Database	Relational Database
Collection Class	Relation
Object	Tuple
Attribute	Column
Method	Procedure

Table 2.1: Object-oriented Database and Relational Database

database-independent since they can be written in the same objected-oriented programming language, while stored procedures are not database-independent due to different database vendors have different stored procedure languages.

However, OODBs rarely perform well in dealing with queries which require significant use of traditional data. Traditional data, such as integer, char, string and boolean, are very simple and object-oriented model is designed to support complex data structures. Therefore, if lots of traditional data are stored as objects in OODBs, a lot of additional information has to be stored as well and this causes performance problems compared with relational databases. Another disadvantage of OODBs is that it lacks a common data model and standards.

2.6 Blob Data Storage

Generally, there are two approaches to store large objects(BLOBs): storing in a file system and storing in a database. The decision is based on the size of blobs, the file system, the

workload etc. Some studies show that SQL Sever is more efficient when the blobs are smaller than 256KB, while blobs larger than 1MB are more efficient managed by NTFS [38]. However, both of these two approaches have problems managing a massive number of photos. Facebook has more than 20 billion photographs on their website. Facebook generates and stores four images of different sizes for each uploaded photograph. If each image is stored as a file, there are 80 billion files and more than 20 TB of metadata which is created by the file system. These massive amount of metadata have far exceeded the caching abilities of a system and this causes additional I/O operations on these metadata when reading and writing photographs.

In order to overcome this problem, Facebook develop a new photo storage system, called Haystack [4], to store more than 20 billion photographs on their website. Haystack stores a lot of photos together as a large log structured (append only) object (usually 10GB) and uses the offset of each photo to retrieve the photo in the corresponding object. There are only 6 million objects in the file system. In this way, Haystack greatly reduces the amount of metadata and provides high disk read throughput. However, Haystack still has some limitations:

1. Lack of Fault Tolerance: Haystack uses RAID-6 to provide high read performance and fault tolerance for disk failure. However, in case that the sever crashes, Haystack cannot respond to the requests for the data on the crashed sever.

2. Slow Index File Recovery: If the sever crashes, the index file in Haystack has to be rebuilt from the haystack file and this is extremely expensive.

3. Compaction Operation: The compaction operation is used to reclaim the space by

the deleted photos by copying the haystack while skipping the deleted photos. However, it is very expensive because it has to create a new copy of haystack. It causes problems if requests come at the same time.

4. No Capacity Balancing: The volume id is hardcoded in the photo and this leads a problem when the haystacks need to be moved for capacity balancing.

We will build a new object store system on Hadoop, called HadoopObS, which will overcome these limitations in Chapter 5.

Chapter 3

System Architecture

In social networks, a large amount of multimedia data such as photo, audio, and video are published and shared by users. These data are so different with nonblob data which are numerals, strings, boolean that we cannot manage it as nonblob data. Typically, blob data

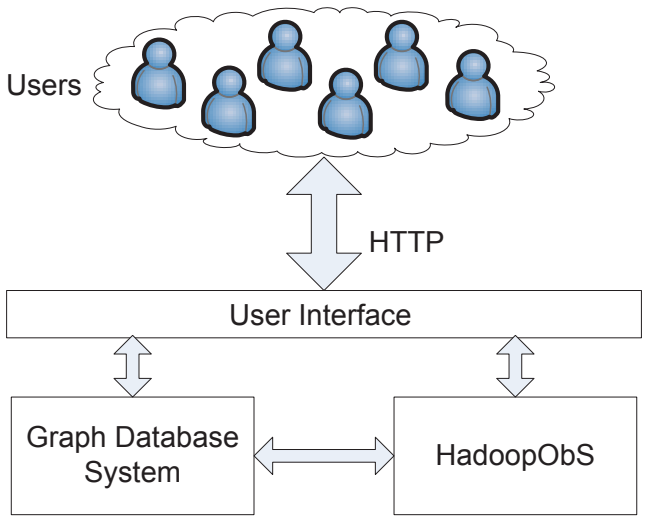

Figure 3.1: System Architecture.

are large objects, such as an image is about 3 MB while a video is even much larger and up to hundreds of MB. Usually, most operations performed on blob data are read operations. Consequently, it is very important to provide a high read speed. As a result, we store blob data apart from nonblob data in an object store system which can provide a high access speed. That is, we divide the data storage problem into two subproblems: nonblob data storage and blob data storage. Nonblob data is stored in a graph database system which will be introduced in Section 3.1, while blob data is stored in an object store system introduced in Section 3.2. The architecture of our system is shown in Figure 3.1.

3.1 Graph Database System

We design a graph database system to manage nonblob data for social network services. In 1977 Leinhardt first introduced the idea of using a directed graph to represent a social community[35]. In Chapter 4, we propose a graph data model, data storage and indexes for the graph database which we design to support social network services.

3.2 Hadoop Object Store

We combine the object store technique and Hadoop Distributed File System (HDFS) to build an object store system on HDFS [3], called Hadoop Object Store(HadoopObS), to store photos for our system and the architecture of HadoopObS is shown in Figure 3.2. HDFS is designed to reliably store very large files across machines in a large cluster. We utilize the features of HDFS, such as replication and cluster rebalancing, to solve the limita-

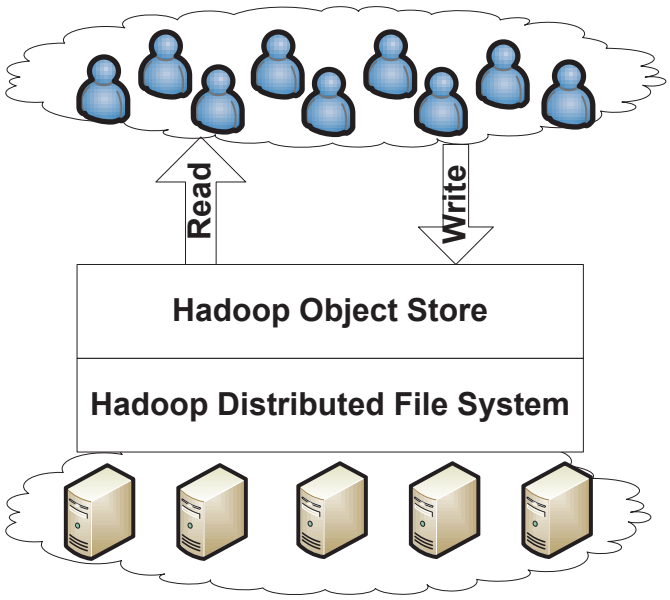

Figure 3.2: The Architecture of HadoopObS.

tions that Haystack suffers. HadoopObS is designed to manage blob data for social network services. We will introduce HadoopObS in Chapter 5.

Chapter 4

Graph Database System

In this chapter, we focus on the nonblob data storage problem. We propose a graph data model which is directed graphs, data storage and indexes for the nonblob data of social network services. Typically, social network graphs are extremely large. Consequently, we also introduce two data partition methods, the Ordering partition method and the MST partition method, to partition the large graphs.

4.1 Graph Model

In this section, we will describe our graph model briefly before we introduce our storage design. Graph models are more natural in representing world facts and beside the data information, structural information is aslo well represented in graph models. Data objects and relationships are typically considered as at the same level in graph models where data objects are nodes and relationships are edges. Therefore, we introduce our graph model in

two aspects: nodes and edges.

In our graph model, there are two types of the nodes, user nodes and object nodes which are published by users and can be photos, blogs, videos and so forth. In social networks, users are always the most important entities and play significantly different roles from other entities. As a result, we classify the nodes of the graph model into two categories and the definitions of them are as following:

Definition 4.1.1 *A user node U is a virtual person in the social network who enjoys their rights and performs their obligations.*

Definition 4.1.2 *A object node O is a form of information or content which is published or shared among users and owned by the user who published it.*

The relationships in social network are extremely complicated and these relationships can be classified into three categories: user-user relationships which connect two users, user-object relationships which connect a user and an object, and object-object relationships which connect two objects. These relationships are represented by labeled edges which specify the attributes of each relationship. For instance, a tagging relationship is one of user-object relationships which can be defined as following:

Definition 4.1.3 *A tagging relationship* $U \xrightarrow{T} O$ *represents a user behavior that a user U tags an object O using a tag* $T = \{c, t, ...\}$*, where c is the content of the tag T, t is a timestamp and T may also contain other related information. The corresponding graph model is shown in Figure 4.1.*

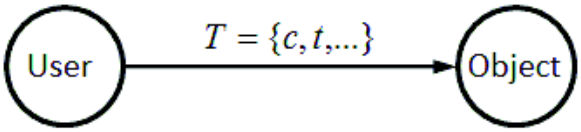

Figure 4.1: The Tagging Relationship in the Graph Model.

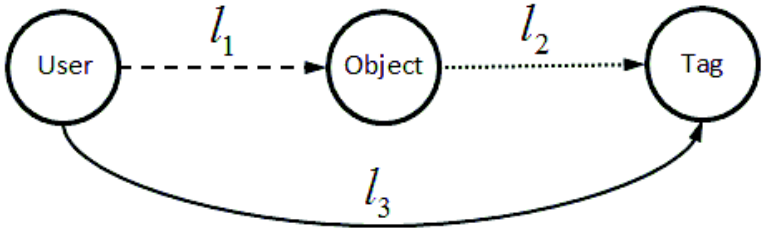

Figure 4.2: Another kind of representation for tagging relationship in the graph model. The three types of lines indicate three different types of relationships and the labels (l_1, l_2, l_3) define the type for each edge respectively.

On the other hand, we can model a tag as a node instead of modeling it as an edge. If a tag is modeled as an node, we have three edges to represent the relationships among these three nodes: a user node, an object node and a tag node as shown in Figure 4.2. Each edge is labeled using a symbol which specifies the type of the edge. The first type of models is suitable for modeling relationships which are simple data structures, for instance, a tag is usually a word or several words. On the other hand, the second type of models is appropriate for modeling relationships which are complex data structures, such as a comment can contain hundreds of words and even some images. This is also the reason we introduce two types of models both supported in the graph database.

4.2 Data Storage

A graph consists two types of elements: nodes and edges, therefore, the problem of data storage for the graph model is divided into two subproblems: node storage and edge storage. In Sec. 4.1, we describe two types of models for the tagging relationship in our graph model. Correspondingly, we introduce two types of storage formats in our storage system.

In the first model, we model tags as edges. In our graph model, nodes are used to model the entities in social network services which can be quite complex, as a result, we store them as objects which can provide more complex data structures than tables. In addition, entities can have independent identification and existence by modeling and storing as objects. Compared with nodes, edges are usually much more simple and storing edges as tables can improve the access speed. We store each type of nodes as a collection of object instances and each type of edges as a table in which each tuple is an edge of the graph as Figure 4.3 shows.

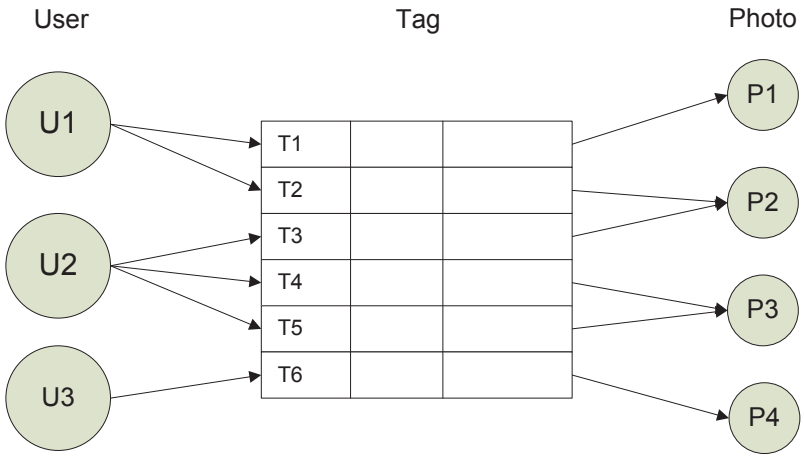

Figure 4.3: Storage Format for the Graph Model Described in Figure 4.1.

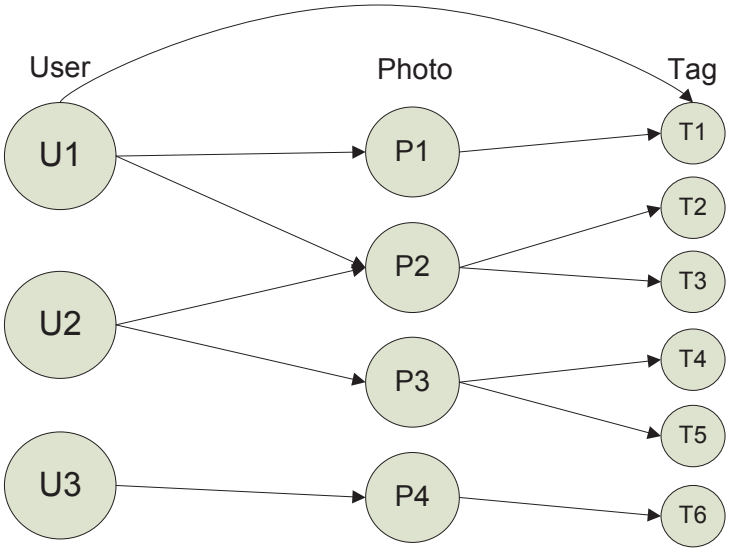

Figure 4.4: Another Storage Format for the Graph Model Described in Figure 4.2. Only one of the edges between users and tags is given to illustrate the relationship between users and tags while other edges are ignored.

In the other model, we model tags as nodes instead of edges and the corresponding storage format is indicated in Figure 4.4. Due to tags are modeled as nodes, tags can support complex data structures or special functions such as users define their own attributes of their tags. The labeled edges between User node, Photo node and Tag node contain no information and just link the nodes together. Therefore, we store them as inverted lists as

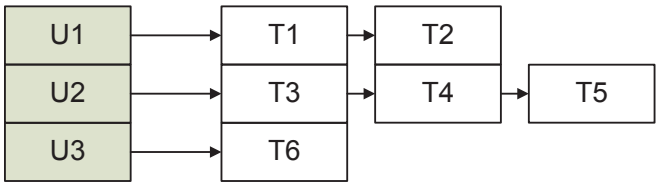

Figure 4.5: A Sample of Inverted List.

Figure 4.5 shows. These inverted lists are used to link the nodes in the graph, while a join index is a binary relation[43]. A join index makes the two joined tables smaller to speed up the join operation. These inverted lists in our graph model are different with the join indexes.

The graph database system has both of the two types of models which make users flexible to model a relationship as an edge or a node. For instance, the relationships have complex data structures, such as the comment relationship, are modeled as nodes, while the relationships without complex data structures, such as the tagging relationship, are modeled as edges.

4.3 Data Partition

Graph models are flexible in modeling complex data models and representing structural information of complex data models. In social networks, structural information is really important and used to detect communities, process queries and etc. In addition, social networks are huge graphs, for example, there are millions of users and billions of photographs on Flickr. It is obvious that one machine has problems in handling these huge graphs. We have to partition them into many small graphs and distribute these small graphs over a cluster of servers.

One of the major properties of graph models is that graph models represent structural information well. Typically, a digraph $G(V, E)$ consists the following structural components:

1. Isolated Node

   An isolated node v of graph $G(V, E)$ is a simple node such that both the in-degree and out-degree of v are 0 where $v \in V$.

2. Isolated Subgraph

   An isolated subgraph $G^{'}(V^{'}, E^{'})$ of graph $G(V, E)$ is a simple subgraph such that no interior node of $G^{'}$ is connected to any interior node of $G^{''}(V^{''}, E^{''})$, where $G^{''}$ is complement of $G^{'}$ over G.

We utilize the structural information to partition social network graphs into small graphs and distribute them over a cluster of servers. A social network graph is a huge graph and contains several types of nodes. Different types of nodes contain different content and perform different roles in social networks.

Therefore, both nodes and edges in our graph model are firstly divided into collections according to the types. Then each collection of nodes is divided into small collections called families by clustering or ordering, while each collection of edges is horizontally partitioned into groups of records called tablets. Finally, the objects are stored according to the families while the tuples are stored according to the tablets.

4.3.1 Ordering Partition

Ordering partition is a popular partition method and used in many systems, such as Bigtable and PNUTS. Typically, ordering partition divides data items according to the lexicographic order of key values. However, the relationships in social networks are extremely complicated as a result there are a large number of edges which form a complex graph structures.

In order to efficiently manage these edges, we define one type of these edges as a primary type of edges for each type of nodes, called *primary relationship*. The primary relationship is one type of the most important relationships for the nodes and, as Figure 4.6 shows, an

edge between U_i and P_j indicates a user U_i uploaded a photo P_j and this is the primary relationship of *Photo* nodes. We order the objects according to the primary relationship and this makes the edges of this relationship clustered as shown in Figure 4.6, while if we order the objects according to the lexicographic order of the key value, the edges will not be clustered as Figure 4.7.

Clustered edge partition can greatly improve the performance of queries. For example, when retrieving a photo's all tags, all tags of a photo is continuously stored because the relationships between photographs and tags are primary relationships. This method greatly reduces the random I/O cost and then improves the performance.

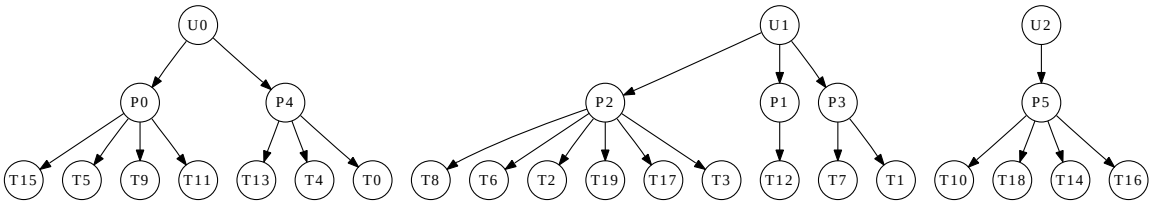

Figure 4.6: Ordering According to the Primary Relationship. The edges between U_i and P_j denote U_i uploaded P_j and the edges between P_i and T_j indicate P_i is tagged by T_j.

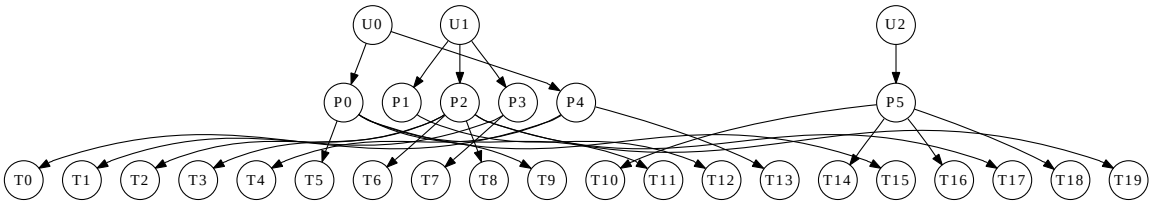

Figure 4.7: Ordering According to the Lexicographic Order On the Key Value.

4.3.2 Minimum Spanning Tree Partition

In social networks, one of fundamental problems is the discovery of clusters or communities. Typically social network data consists lots of interaction information among the users. We calculate the distances of users based on these interaction information. Suppose all applications in a social network are divided into m categories, such as blog, album, game and so on. Correspondingly, all interactions are divided into m categories $I_1, ..., I_m$ and N_i is the number of I_i interactions. The weight of I_i interactions is

$$W_i = \frac{N_i}{\sum_{j=1}^{m} N_j} \tag{4.1}$$

Then we can calculate the length of the edge between two users, and the length of the edge between u_i and u_j is

$$L(u_i, u_j) = \frac{1}{\sum_{k=1}^{m} W_k n_k} \tag{4.2}$$

where n_i is the number of I_i interactions between u_i and u_j. Therefore, we obtain a weighted graph in this social network, in which each node is a user and each weighted edge is the distance between two connected users. Usually, clustering algorithms, such as K-Means clustering [34] and Spectral clustering [27], can be used to partition this kind of weighted graphs. Unfortunately, the time complexity needed to achieve this is extraordinarily high. For instance, if we define the distance for a pair of users in the graph,

$$D(u_i, u_j) = ShortestPath(u_i, uj) \tag{4.3}$$

then we define a distance matrix $M \in \Re^{n \times n}$ where n is the number of users and $m_{ij} = m_{ji} = M[i][j] = D(u_i, u_j)$. In order to obtain the distance matrix M, we have to calculate the shortest paths for all pairs in the graph. This is an all-pair shortest-paths problem, and the

time complexity needed to solve this problem is $\Theta(V^3 \lg V)$. For a graph with billions of vertexes, it is impossible to be handled over this time complexity.

Consequently, instead of clustering vertexes, we construct minimum spanning trees on the graphs. The minimum spanning tree problem can be easily solved in time $O(E \lg V)$, such as Kruskal's algorithm. If using a highly parallelized manner with a linear number of processors, this problem can be solved in time $O(\lg V)$[14]. Consequently, instead of clustering the nodes, we construct minimum spanning trees on the graphs and then partition the nodes.

Algorithm 1: WGraph(U, I)

Input: $U = \{u_1, u_2, ...\}$ (U is a set of users)

$I = \{I_1, I_2, ...\}$ (I is a set of all interactions among the users and $I_i \in I$ is a category of interactions)

Output: $G(V, E, W)$ (G is a social network graph)

```
1 V = U;
2 foreach I(v_i, v_j) ∈ I do
3     if i < j then
4         e = (v_i, v_j);
5         w = 1 / (Σ_{k=1}^{m} W_k n_k);
6         Add e to E;
7         Add (e, w) to W;
8 return G(V, E, W);
```

We use Algorithm 1 to construct the weighted social network graph. Then we use Kruskal's algorithm [30] to build the minimum spanning tree on this graph. In Algorithm 1, w_k in

$w = \frac{1}{\sum_{k=1}^{m} W_k n_k}$ is calculated using Equation 4.1.

Algorithm 2: MSTPartition($G(V, E, W), n$)

Input: $G(V, E, W)$ (G is a social network graph)

n (n is the number of partitions)

Output: $P = \{P_1, P_2, ..., P_n\}$ (P is the set of partitions and each P_i is a partition)

```
1  T = KruskalMST(G, E, W);
2  Q = BFS(T);
3  i = 1;
4  P = {P_1, P_2, ..., P_n};
5  foreach P_i ∈ P do
6      P_i = ∅;
7  foreach v ∈ Q do
8      Add v to P_i;
9      if |P_i| > ⌈|V|/n⌉ then
10         i = i + 1;
11 return P;
```

In Algorithm 2, we use Kruskal's algorithm to construct a minimum spanning tree. Then, the breadth first search(BFS) algorithm is used to search the minimum spanning tree which we have constructed. After this, we obtain a queue of the nodes according to the order of the nodes searched in the BFS algorithm. Finally, we partition the nodes in the queue into n groups.

4.4 Indexes

Indexing is the most important and fastest approach which reduces high I/O cost effectively and greatly improves the speed of data retrieval. Furthermore, social network sites have a massive amount of data. Consequently, the responsibilities and roles of indexes in data retrieval for social network services are significant. In this section, we introduce two types of indexes: content index and node index.

4.4.1 Content Index

Content indexes are build on the attributes of nodes and edges to support keyword search. It is implemented as B+-tree index is used to support keyword search as shown in Figure 4.8. If the keyword search is $KS = \{w_1, w_2, ...\}$ where each w_i is a word, we use the content

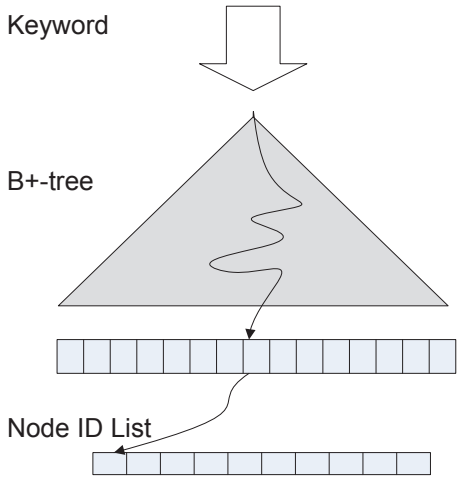

Figure 4.8: Content Index.

index to obtain a *Node ID List* for each word $w_i \in KS$. Then a merge join is performed on

all of the lists and the final result is gained. On the other hand, if given a keyword search $KS = \{w_1\ or\ w_2\ or\ ...\}$, we use the content index to gain a *Node ID List* for each $w_i \in KS$ and then merge all the lists.

4.4.2 Node Index

Our node index is similar with content index but build on node identities. Recall that in our graph, we divide nodes into two categories: User nodes and Object nodes, therefore, there two types of node indexes build on these two categories of nodes are different as shown in Figure 4.9 and 4.10. For a given *User ID*, the *Object ID Lists* are all objects which

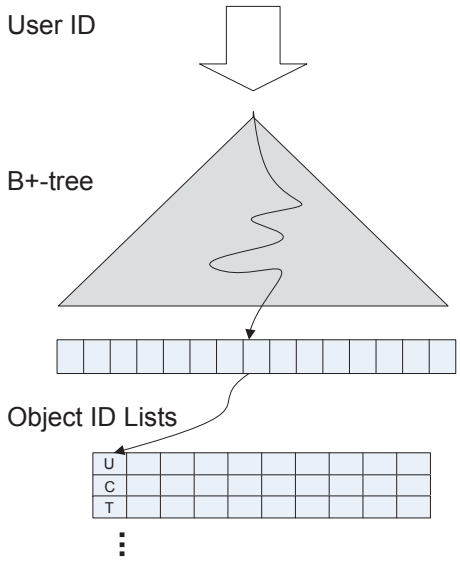

Figure 4.9: User Node Index.

are uploaded, commented or tagged by the users. These objects are classified into different categories according to the types of the user-object relationships, such as uploading, commenting and tagging and each list is labeled to specify the type of the list. However, for a

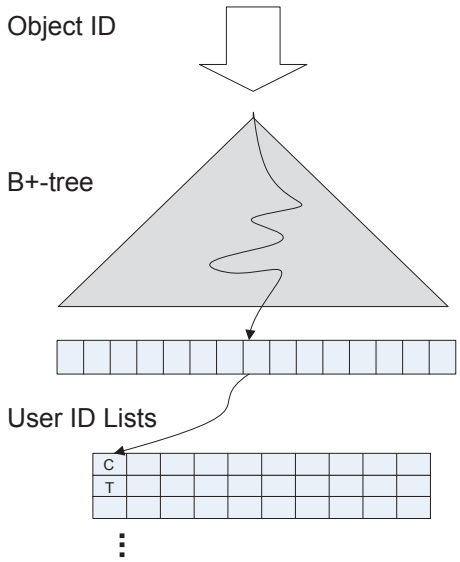

Figure 4.10: Object Node Index.

given *Object ID*, the *User ID Lists* are all users who have relationships with this object and these users are classified into different categories according to the relationships with this object. Each list is labeled to specify the type of the relationships as shown in Figure 4.10.

4.5 Simulation

In this thesis, we have not implemented the graph database which is designed to serve social network services. Therefore, in order to evaluate our storage and index design, we simulate our graph database on the relational database system shown in Figure 4.11. Nodes, edges and indexes are converted to tables stored in relational databases. In relational databases, a foreign key is a referential action which defines a relationship between two tables. This is an indirect connection and if we want to connect two tables, we have to do a join operation

on the two tables. Unfortunately, a join operation is very costly. We use links instead of foreign keys to represent the relationship between two tables and that is, a whole database is a graph. A relationship in the graph model is converted to tables. We take the tagging relationship as an example to explain how this conversion is processed. For a given tagging relationship $U \xrightarrow{T} O$ or $U \longrightarrow T \longrightarrow O$, the users U are converted to a table $User$ with primary key Uid and the objects O are converted to a table $Object$ with primary key Oid. Then the tags T are converted to a table Tag with primary key Tid, and two foreign keys $TUid$ and $TOid$ which reference from $User$ and $Object$ respectively. A query is divided into several subqueries and passes them to the relational database. After the processor obtains the results from the relational database, it process and merges the results to gain the final result.

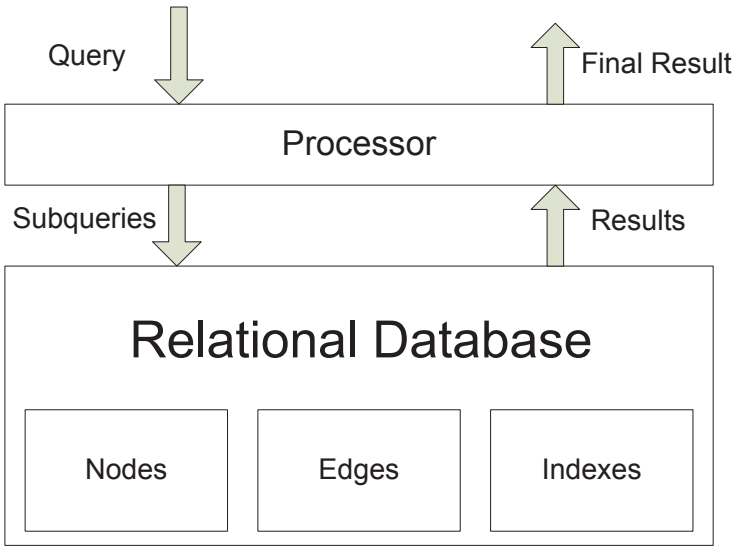

Figure 4.11: Simulation on Relational Database

Chapter 5

HadoopObS

We solve the blob data storage problem in this chapter. In social network services, there are a massive amount of blob data and a wide variety of applications which make frequent file reads on these blob data. Due to most of the operations are read operations, it is the most important thing to improve the performance of read operations. Consequently, we propose our HadoopObS system which is read-optimized system to support these read-intensive applications.

5.1 Metadata and Index

HadoopObS stores a large number of photos together as a large object instead of storing each photo in its own file. Each object is a append-only file and photos are stored in an object until the size of this object reaches the maximum size. An object whose size reaches the maximum size is called a "*full*" object. This greatly reduces the number of files in

HadoopObS and makes the size of total metadata much smaller. This makes it is possible to cache all the metadata of the objects. For example, Facebook has 80 billion image files and more than 20 TB of metadata which is created by the file system. If the photos are stored together as large objects and each object is 10 GB, there are only 6 million objects and 2 GB of metadata in the file system.

On the other hand, HadoopObS also has to maintain metadata of each photo in order to make these photos retrievable. However, traditional file systems are governed by the POSIX standard, and manage metadata and access methods for each file. The metadata in traditional file systems contains lots of information as Figure 5.1 shows, however, only the top three information, file length, device id and storage block pointers, is cared by HadoopObS. More information in metadata makes the metadata too large to be cached and leads to addi-

File length
Device ID
Storage block pointers
File owner
Group owner
Access rights on each assignment: read, write and execute
Time of the last change
Time of the last access
Time of the last modification
Reference counts

Figure 5.1: Metadata in Traditional POSIX File Systems.

tional I/O operations. Consequently, HadoopObS maintains simpler metadata which only

contains the object identifier where the photo is stored, the size, the offset and the flag for each photo and these metadata is stored both in memory and the database system.

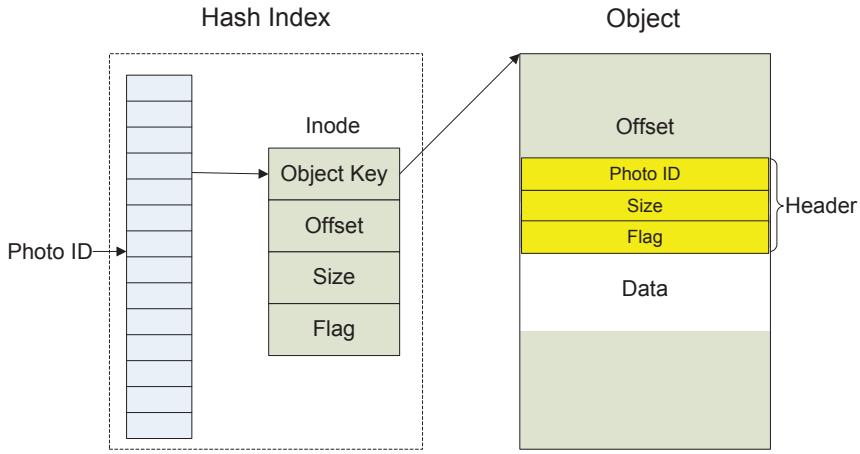

Figure 5.2: Hash Index and Object in HadoopObS.

In memory, the metadata is maintained in a hash index as shown in Figure 5.2. HadoopObS can quickly locate a photo by the given photo id and does not need additional I/O operations. However, memory is not a permanent storage device and all information will be lost if the system crashes. Consequently, in order to provide the reliability for the meatdata storage, HadoopObS also uses the database system to store the metadata. In case the system crashes, the metadata stored in the database system is used to rebuild the in-memory hash index when the system recoveries.

5.2 Operations

In HadoopObS, five operations are defined: read, write, delete, modify and compact operation. The compact operation is a system operation which will be issued by the system

itself or the administrator of the system, while other operations are user operations. Each operation is processed as following in the system.

Read Operation

When a user tries to retrieve a photo, the request is forwarded to the graph database system. The database system finds the information of this photo, including the owner, comments and tags, then passes the photo id to HadoopObS. After HadoopObS receives the photo id, it locates the photo using the in-memory hash index, read the photo data and returns the photo. The process steps of a read operation are shown in Figure 5.3.

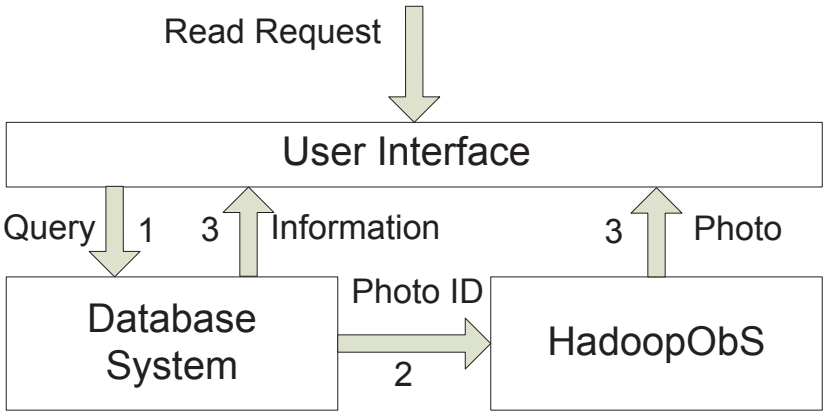

Figure 5.3: The Processing of Read Operation.

Write Operation

When a photo is uploaded, the graph database system inserts the photo's information into the database and passed the photo id to HadoopObS. HadoopObS stores the photo, and updates the in-memory hash index. After finishing these, it passes the metadata (including the photo id, the size, the object id, the offset and etc) to the graph database system. Finally, the graph database system inserts this metadata into the database as shown in Figure 5.4.

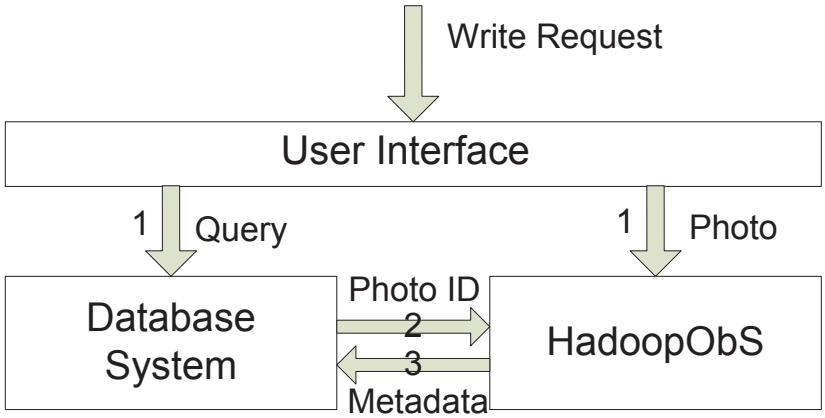

Figure 5.4: The Processing of Write Operation.

Delete Operation

Actually, HadoopObS does not delete the photo. Instead, it updates the in-memory hash index and sets the photo flag to zero to signifying the particular photo has been deleted, while the graph database system updates the metadata of this photo and sets the flag of this photo to be zero.

Modify Operation

HadoopObS supports the modify operation by dividing this operation into one delete operation and one write operation. This operation is necessary because there are some applications which allow users to edit photos, such as color balancing, cropping, and red-eye correction.

Compact Operation

When a photo is deleted, HadoopObS still stores this photo on the disk. If there are many deleted photos, this will waste a lot of disk space and cause the system to be inefficient. Therefore, HadoopObS supports the compact operation to delete these photos from the disk

by copying the object to another object. If a photo flag is zero, it is not be copied to the new object. When this is finished, the system deletes the file of the original object from HDFS. This is an operation which will be issued by the administrator of the system or the system itself.

5.3 NameNode, DataNode and QueryNode

In HadoopObS, there are three types of nodes: NameNode, DataNode and QueryNode as Figure 5.5 shows. The HDFS has one NameNode which manages the file system and a

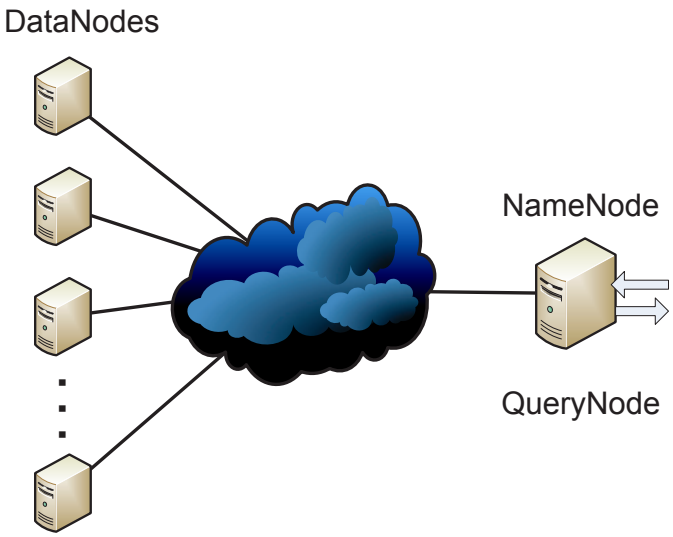

Figure 5.5: The Architecture of the System with One QueryNode.

number of DataNodes. In HadoopObS, we define another type of nodes which are used to process and respond the requests, called QueryNodes. Both the NameNode and DataNodes can act as QueryNodes. For instance, in Figure 5.5, the NameNode acts as a QueryNode.

Consequently, there is at least one QueryNode in a cluster and the number of QueryNodes is flexible.

5.4 Replication and Fault Tolerance

5.4.1 Replication

HadoopObS replicates its files in HDFS on multiple nodes in a cluster to achieve high availability and durability. This replication which is built on top of HDFS replication not only improves availability, but also improves the performance of the system. When a read request comes, the system chooses the replica which is closest to the reader to respond the request.

Besides, if there is no replication, it has a problem when a request is coming while the object is compacted. This problem can be solved by replicating the data. The compact operation is only conducted on full objects and the system only locks one replica when the object is compacted. Therefore, the coming request cannot be a write request or a compact request. That is, the request may be one of the three types of requests which are read requests, delete requests and modify requests. If it is a read request, other replicas which are not locked can respond the read request. Recall that the system does not delete the photo and it sets the photo flag to zero to signifying the particular photo has been deleted. Consequently, if the request is a delete request or modify request, the system does not need to performance any operation on the compacting object. After finishing compacting the object, the system releases the lock and deletes the original object.

5.4.2 Failure Detection and Recovery

In HDFS, each DataNode periodically sends a message to the NameNode. The NameNode detects the failure which may cause by a DataNode failure or a network partition by the absence of messages. When a failed DataNode recoveries, it reads the metadata from the database system instead of scanning all the objects on the node. It is a more efficient way to rebuild in-memory hash indexes.

Chapter 6

Experiment and Evaluation

In this chapter, we conducts a series of experiments to evaluate the performance of the graph database which manages nonblob data and HadoopObS which stores blob data. In Section 6.1, several experiments are conducted to evaluate our storage and index design for the graph database, while we evaluate our HadoopObS in Section 6.2 which contains both single-query experiments and multi-query experiments. Finally, we conduct experiments to evaluate the scalability of the entire system which contains both nonblob data and blob data.

6.1 Nonblob Data Evaluation

6.1.1 Experiment Setup

We conduct our experiments on a computer with an Intel(R) Core(TM) 3.0GHz CPU, 4GB RAM and a 250GB SATA Harddisk running 32-bit Ubuntu Desktop 9.04. In our exper-

# tuples	User	Photo	Comment	Tag
F_1	73250	141160	576174	907550
F_2	145903	308259	1160181	2009748
F_4	292275	838185	2665445	5353712
F_6	426114	1321674	4096295	8227169

Table 6.1: The datasets downloaded from Flickr

iments, the dataset was downloaded form Flickr[2] and stored in four tables as shown in Table 6.1. We use F_1 as a baseline dataset while F_2, F_4 and F_6 are about 2, 4 and 6 times of F_1 respectively. We use five queries to evaluate the conventional method and our graph method with different partition methods, ordering partition method and MST partition method. In the conventional method, we don't utilize the inverted lists and the indexes which we design for our graph database to process queries and all of the five queries have join operations. For example, Q1 is written as

Select Pid From User, Photo Where Uid = PUid and Uname = "Tom"

in SQL in the conventional method and submitted to the database system.

Q1: Given a user name, retrieve all photos of his /hers.

Q2: Given a list of users' names, retrieve all photos of theirs.

Q3: Given a photo id, retrieve all photos of its owner's.

Q4: Given a list of photo ids, retrieve all photos of their owners'.

Q5: Retrieve all users who have uploaded photos but have not published any comments.

We build our content index and node index on the datasets and these indexes cost additional disk space. In Figure 6.1, we compare the dataset sizes with indexes and without indexes. It shows that our index only costs a little more storage space but it can greatly improve the performance for processing some queries.

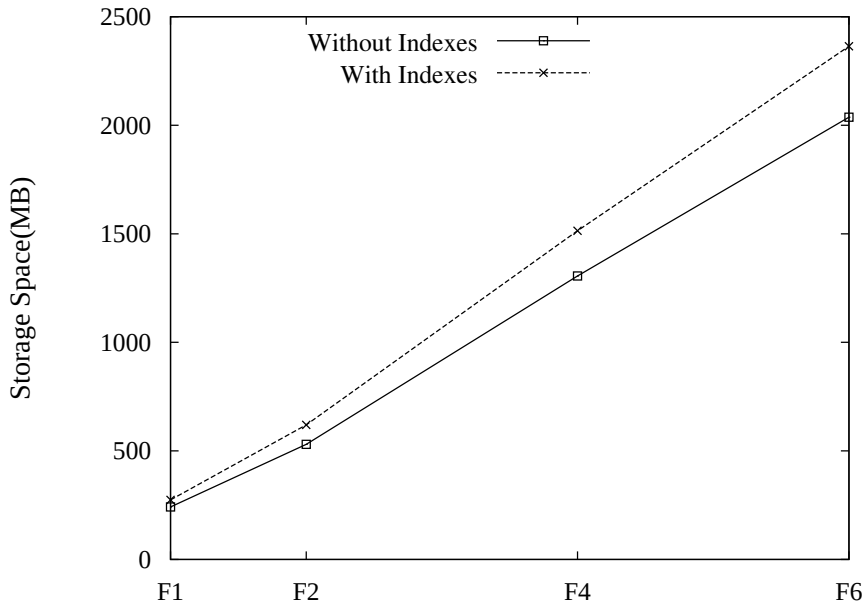

Figure 6.1: Storage Space for Indexes.

6.1.2 Result

We use Q1 and Q2 to evaluate the joins from the referenced table to the referencing table. The results show that both the MST method and the Ordering method outperform the conventional method in both performance and scalability in Q1 and Q2 as shown in Figure 6.2 and Figure 6.3. The query processing time of Q2 in the MST method and Ordering method slightly increases but it largely increases in the Conventional method. In addition,

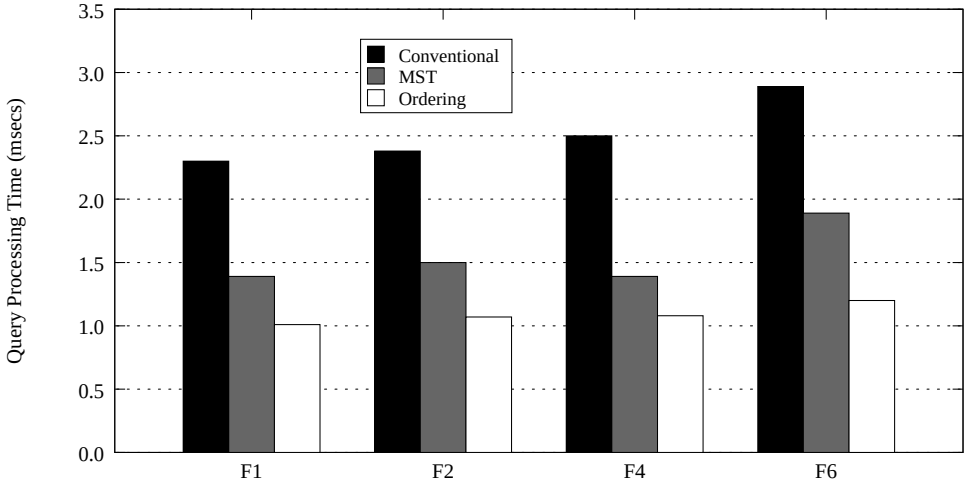

Figure 6.2: Query Processing Time of Q1 (Given a user name, retrieve all photos of his /hers).

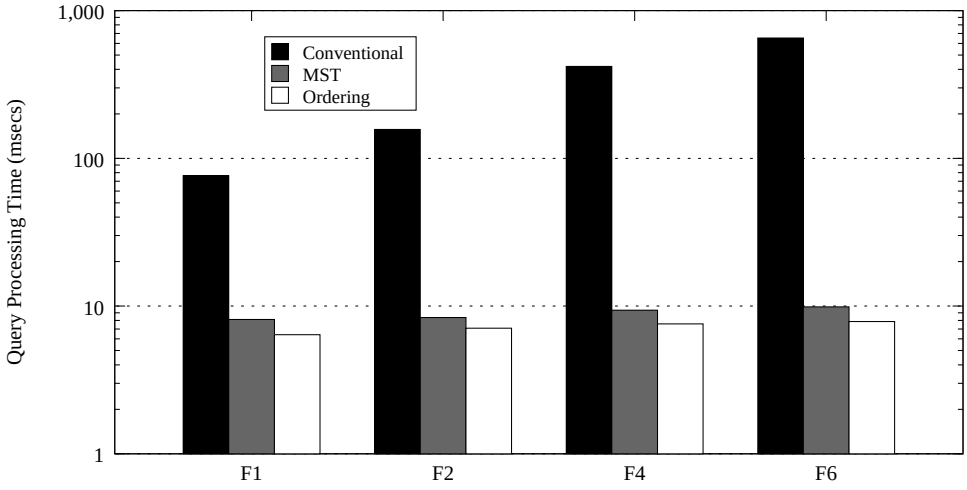

Figure 6.3: Query Processing Time of Q2 (Given a list of users' names, retrieve all photos of theirs).

the Ordering method also outperforms the MST method and this performance improvement is contributed by the clustered edge partition.

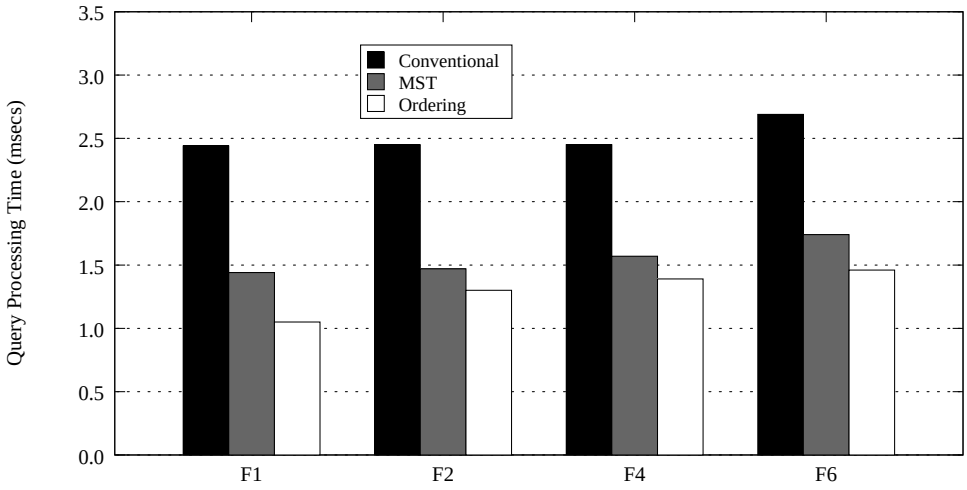

Figure 6.4: Query Processing Time of Q3 (Given a photo id, retrieve all photos of its owner's).

We use Q3 and Q4 to evaluate the joins from the referencing table to the referenced table. Obviously, both the MST method and the Ordering method outperform the conventional method in both performance and scalability, while the Ordering method slightly outperform the MST method as Figure 6.4 and 6.5 show.

Q5 is used to evaluate the performance of processing a query which has two join operations and Figure 6.6 shows that both the Ordering method and MST method outperform the conventional method as well.

We compare the performance of the ordering partition method and the MST partition method by measuring the average time of retrieving all photos of a user and the average time of retrieving all comments and tags of a photo. We randomly choose 1,000 users

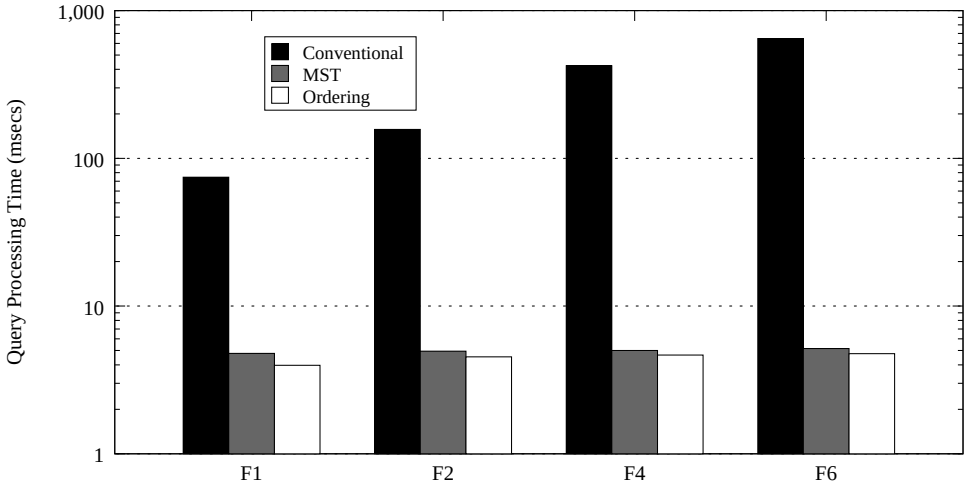

Figure 6.5: Query Processing Time of Q4 (Given a list of photo ids, retrieve all photos of their owners').

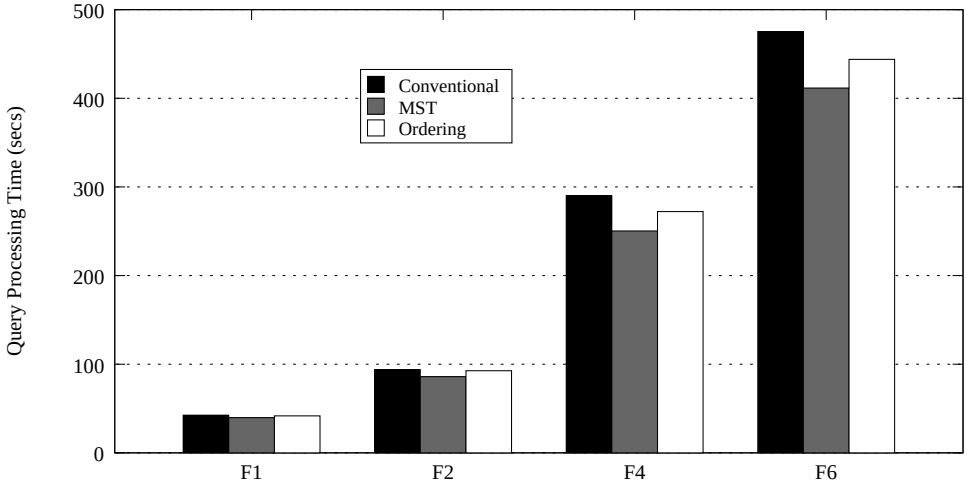

Figure 6.6: Query Processing Time of Q5 (Retrieve all users who have uploaded photos but have not published any comments).

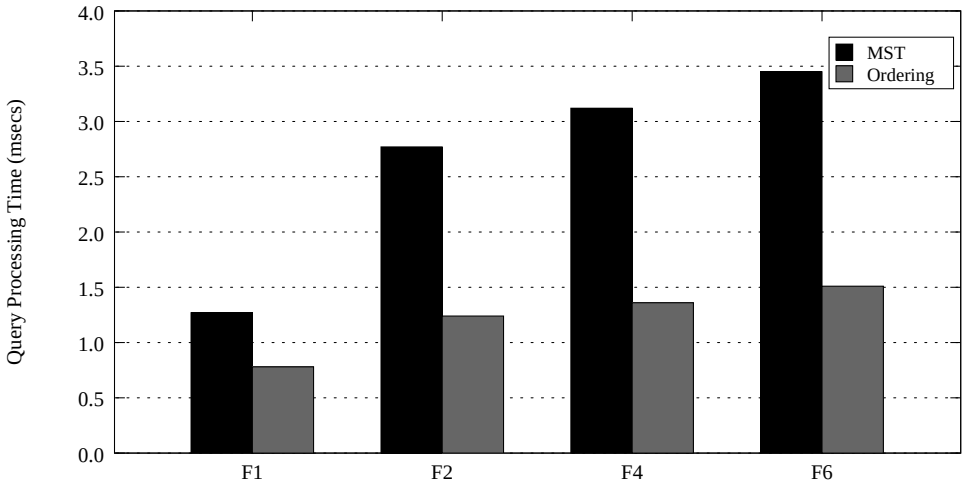

Figure 6.7: Average Time of Retrieving a User's Photo.

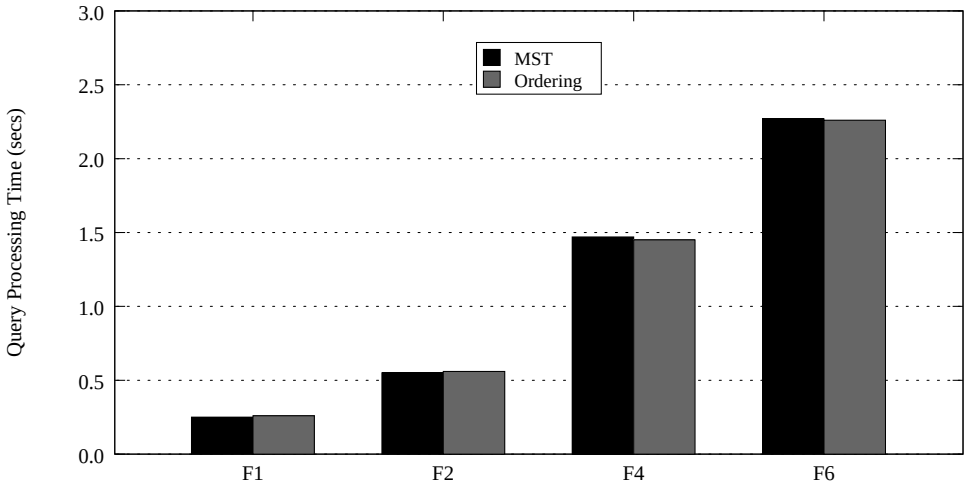

Figure 6.8: Average Time of Retrieving a Photo's Comments and Tags.

to retrieve their photos and calculate the average retrieval time. Figure 6.7 shows that the Ordering method is much faster than the MST method. This is due to in the Ordering method, we define the primary relationships, and partition the nodes according to these relationships. This makes the edges clustered and reduces random I/Os.

Then we randomly choose 10,000 photos to retrieve all comments and tags on them, and calculate the average retrieval time. The result shows that the average time in two methods is almost the same as Figure 6.8 shows.

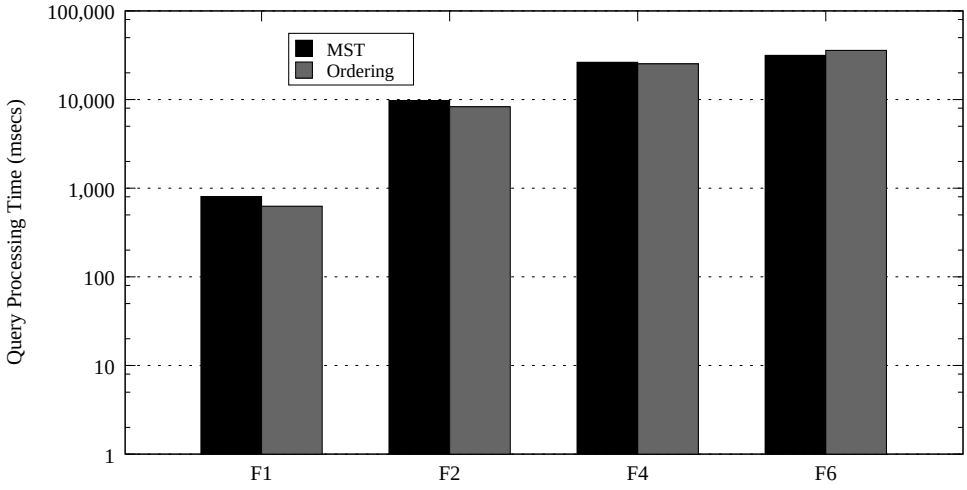

Figure 6.9: Query Processing Time of Retrieving the Latest Comment of Each Photo.

We also run some queries to evaluate these two methods. First, we run a query which aims to retrieve the latest comment of each photo and the result is shown in Figure 6.9. The performance of the two methods is almost the same. Then, a query which aims to retrieve the latest photo of each user is run and the result shows the Ordering method slightly outperforms the MST method shown in Figure 6.10.

These results show that the MST method does not perform as well as the Ordering method.

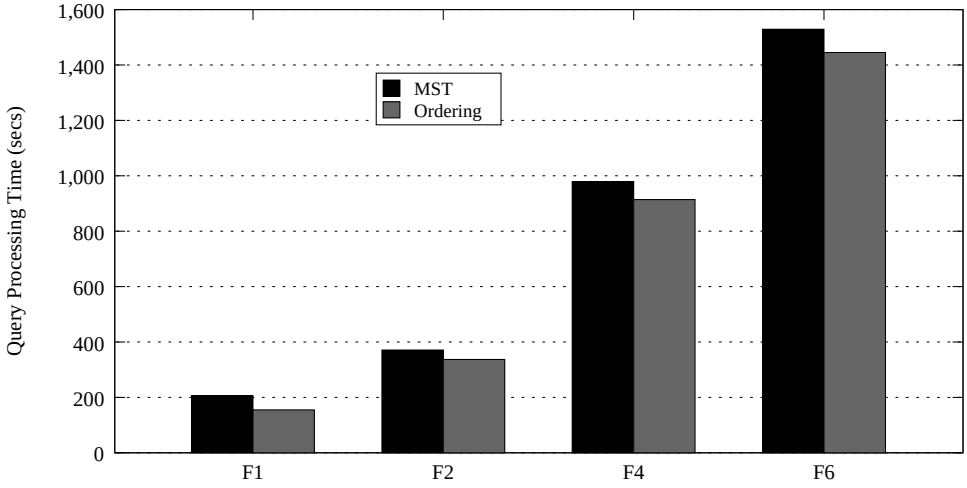

Figure 6.10: Query Processing Time of Retrieving the Latest Photos of Each User.

This is because the operations in relational databases are based on the set theory, while the operations in graph databases should be based on the graph theory. As a result, the Ordering partition method performs better than the MST method in our simulation on relational databases. However, we think the MST method should perform better than the Odering method in graph databases instead of simulation on relation databases.

6.2 Blob Data Evaluation

6.2.1 Experiment Setup

Our experiments are conducted on 14 nodes of our Awan cluster where one node is used as the NameNode and other nodes are used as DataNodes. Each node has an Intel(R) Xeon(R) X3430 Quad Core CPU, 2×4GB memory and 2×500GB SATA II Hardisk, and runs 64-bit platform Linux CentOS. For our experiments, we use Hadoop version 0.19.2 running

on Java 1.6.0. We deployed the system with several changes to the default configuration settings. Data in HDFS is stored using 512MB data blocks instead of the default 64MB.

6.2.2 Single-Query Experiments

We randomly read 50,000 photos and calculate the average time of reading a photo. In HadoopObS, each object is 5GB containing thousands of photos, while in the Smallfile method, we store each photo in its own files in a hierarchical structure. The result shows that HadoopObS outperforms Smallfile more than two times shown in Figure 6.11. When

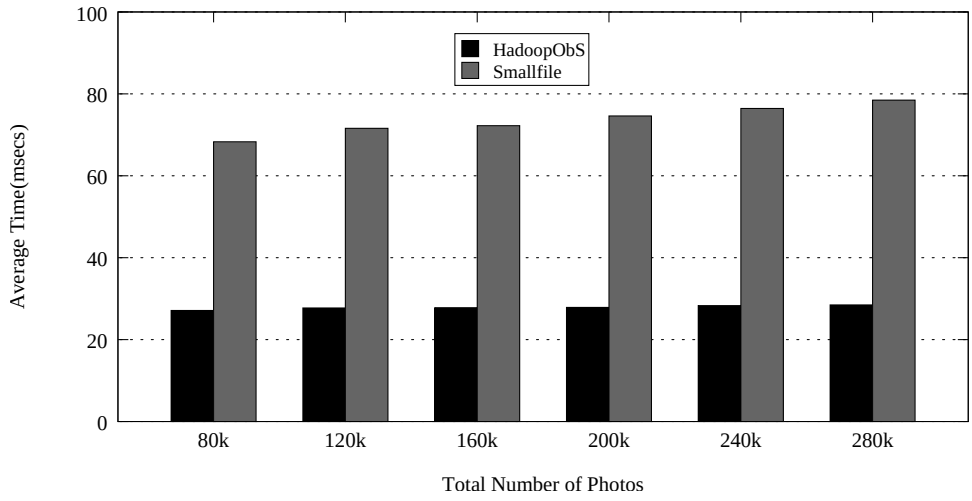

Figure 6.11: Average Time of Reading a Photo.

the number of photos increases, the average time slightly increases in HadoopObS. However, it is obvious that the the average time increases faster in Smallfile than in Bigfile. Therefore, HadoopObS scales better than Smallfile method. This is due to, in Smallfile, the number of files increases faster than HadoopObS and this costs more disks space to store the metadata of the files and more time to locate the target photo. However, HadoopObS

can rapidly find the target photo through the in-memory hash index and does not cause additional I/O operations.

In order to evaluate the write operation in HadoopObS, we write 10,000 photos and calculate the average time of writing a photo compared with the Smallfile method. Figure 6.12 shows that the Smallfile method only slightly outperforms HadoopObS.

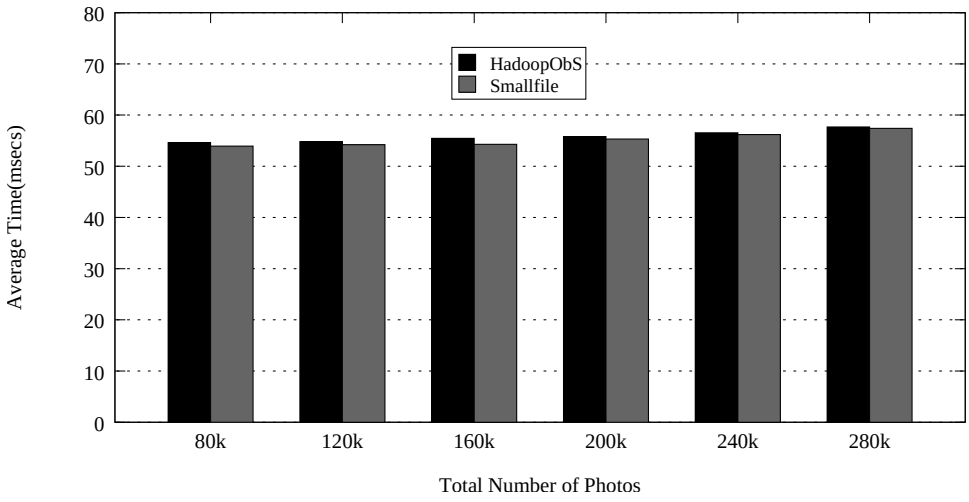

Figure 6.12: Average Time of Writing a Photo.

The compact operation is the most costly operation in HadoopObS. However, the result shows that the average time of compacting an object has a weak linearly increasing when the number of total photos increases linearly shown in Figure 6.13.

We also measure the throughput of reading and writing in HadoopObS and the Smallfile method. Figure 6.14 shows that the read throughput of HadoopObs is about two times of the Smallfile's and this is consistent with the results shown in Figure 6.11. Besides, the read throughput of the Smallfile also decreases faster than HadoopObS's. It is shown that the write throughput of HadoopObs is a little smaller than the Smallfile's in Figure 6.15. This

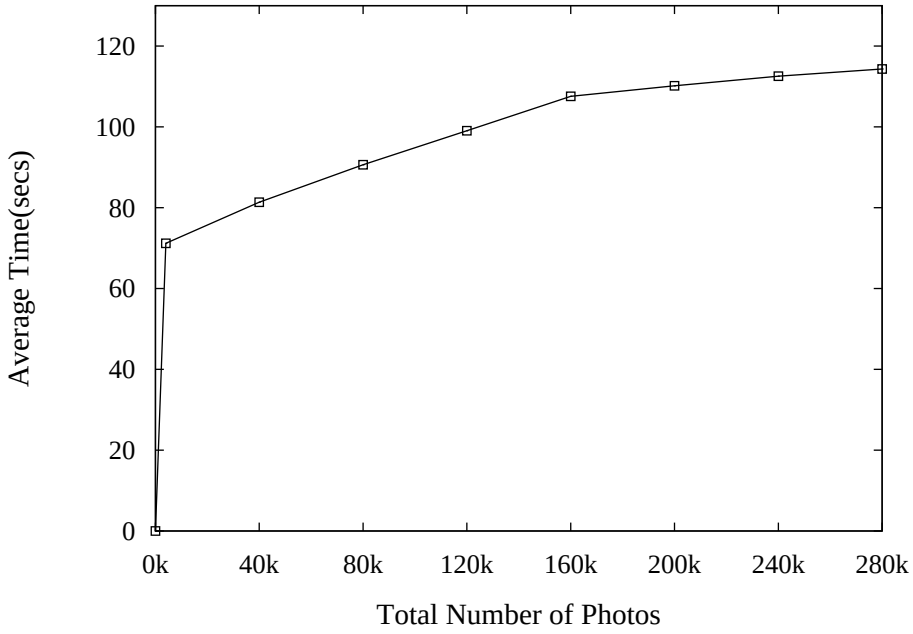

Figure 6.13: Average Time of Compacting an Object.

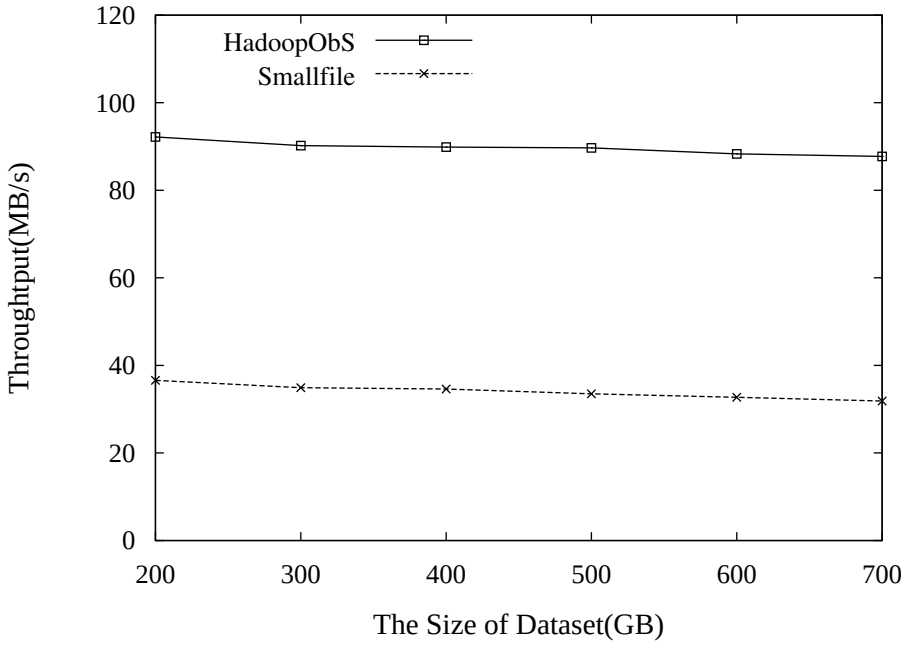

Figure 6.14: The Throughput of Reading.

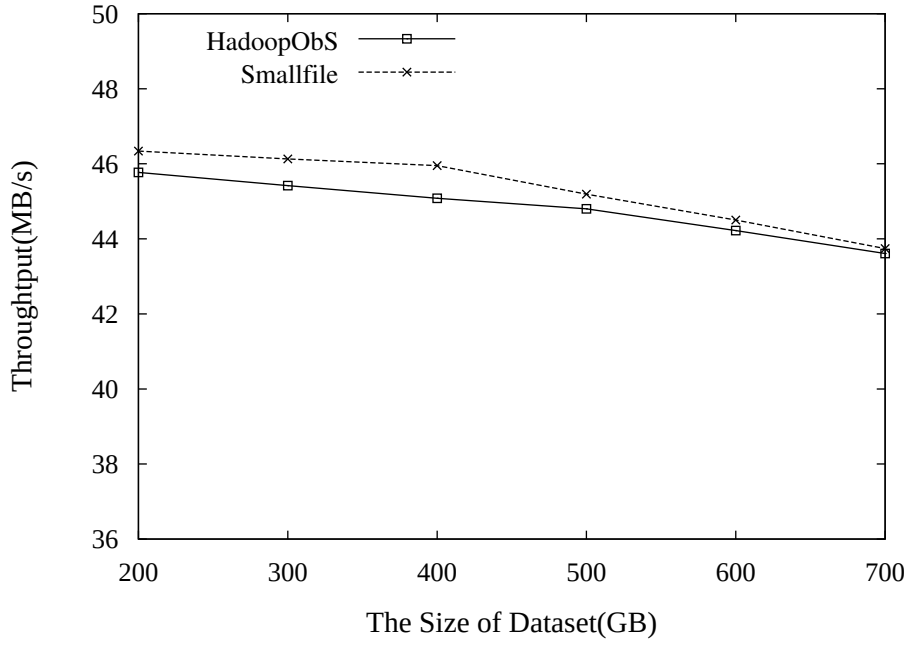

Figure 6.15: The Throughput of Writing.

is because HadoopObS needs to update the hash index and stores the metadata of the new photo. However, the two lines in Figure 6.15 get closer when the total data size increases due to when the number of photos increases, it needs more time to check if the file which will be created has existed in the Smallfile method.

6.2.3 Multi-Query Experiments

Social network services aim to support a massive number of users and have to process a lot of requests submitted by these users every second. Therefore, in this section, we conduct some experiments to evaluate the scalability of HadoopObS when the concurrency increases. We randomly generate a photo id for each request and retrieve the photo according to the given photo id. The maximum transmission rate of the links between the switch

and the nodes are 1 Gbps. The average size of the photos is 2.5 MB and the total number of photos is 280,000.

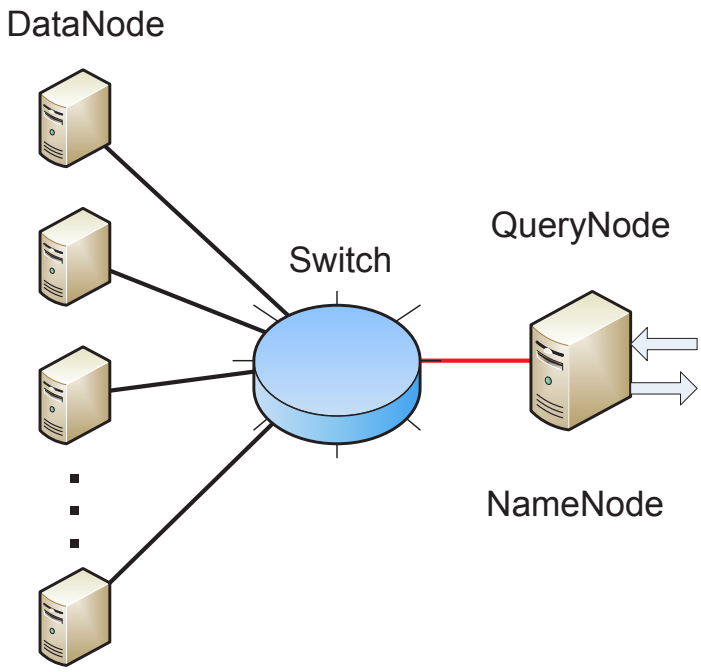

Figure 6.16: The Architecture of the System with One QueryNode.

First, we choose one of the 14 nodes as the QueryNode shown in Figure 6.16. The throughput of the system is measured and the result is shown in Figure 6.17. The maximum throughput T is 40.75 photos/second.

Then, we increase the number of QueryNodes and measure the throughput of the system. Figure 6.18 shows that when the number of QueryNodes increases, the maximum throughput of the system increases sublinearly.

We analyze the maximum throughput of the system when the number of QueryNodes increases. We assume that the bottleneck of the system is the links between the switch and

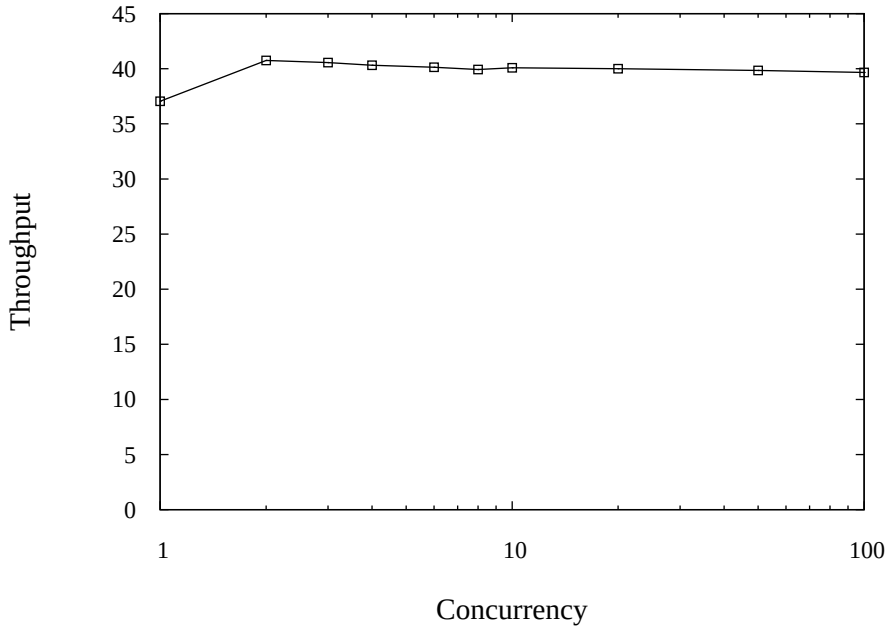

Figure 6.17: The Throughput of the System with One QueryNode.

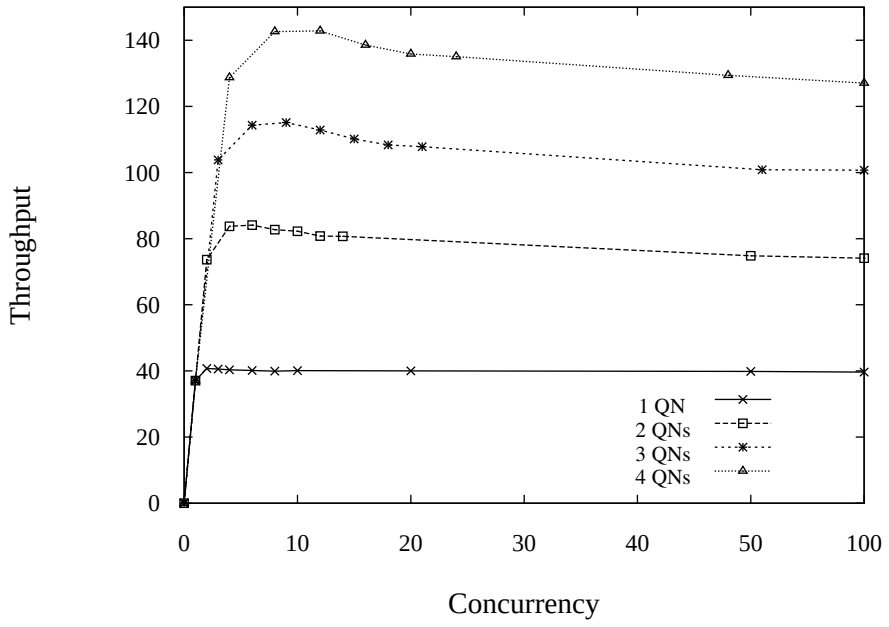

Figure 6.18: The Throughput of the System When the Number of QueryNodes Increases.

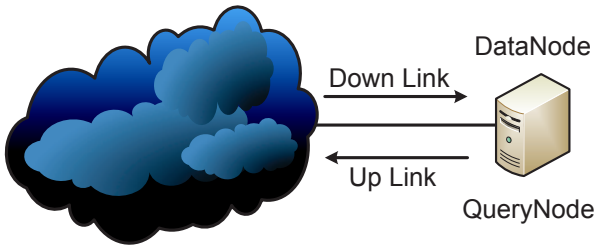

Figure 6.19: The DataNode which acts as a QueryNode.

Symbol	Definition
T	The Max Throughput (photos/second)
Q	The Number of QueryNodes
N	The Number of DataNodes
R	The Maximum Transmission Rate (photos/second) of the Links
R_u	The Maximum Transmission Rate (photos/second) of the Up-Links
R_d	The Maximum Transmission Rate (photos/second) of the Down-Links

Table 6.2: The Definitions of the Symbols

the QueryNodes which also act as DataNodes as shown in Figure 6.19 and define the some symbols in Table 6.2. The throughput and transmission rates in these experiments are measured by photos/second.

The maximum transmission rate of the up-link between the switch and a QueryNode is

$$R_u = \frac{T}{N} - \frac{T}{QN}, \tag{6.1}$$

while the maximum transmission rate of the down-link is

$$R_d = \frac{T}{Q} - \frac{T}{QN}. \tag{6.2}$$

Therefore, we have

$$R = R_u + R_d = \frac{T}{N} - \frac{T}{QN} + \frac{T}{Q} - \frac{T}{QN}. \tag{6.3}$$

Finally, the maximum throughput of the system is

$$T = \frac{RNQ}{Q + N - 2}. \tag{6.4}$$

According to result of the experiment shown in Figure 6.17, R is about 40.75, while $N = 13$. Consequently, the maximum throughput of the system is

$$T \approx \frac{529.75Q}{Q + 11}. \tag{6.5}$$

Then, we conduct experiments to verify this model and the result is shown in Figure 6.20. Figure 6.20 shows that when Q (the number of QueryNodes) ≤ 8, the two lines match each other well. That is, when $Q \leq 8$, the bottle neck of the system is the links between the switch and the DataNodes.

Finally, we measure the throughput of system with all 14 nodes as QueryNodes. The maximum throughput is about 265 when the concurrency is 26. The results of these experiments show that our HadoopObS perform very well on Awan.

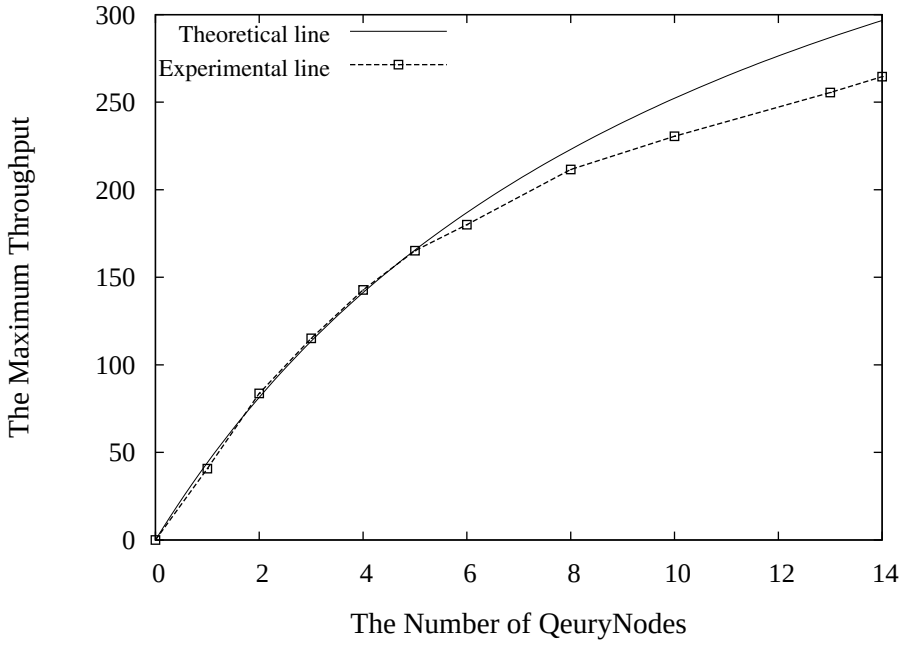

Figure 6.20: The Maximum Throughput with the Number of QueryNodes Increases.

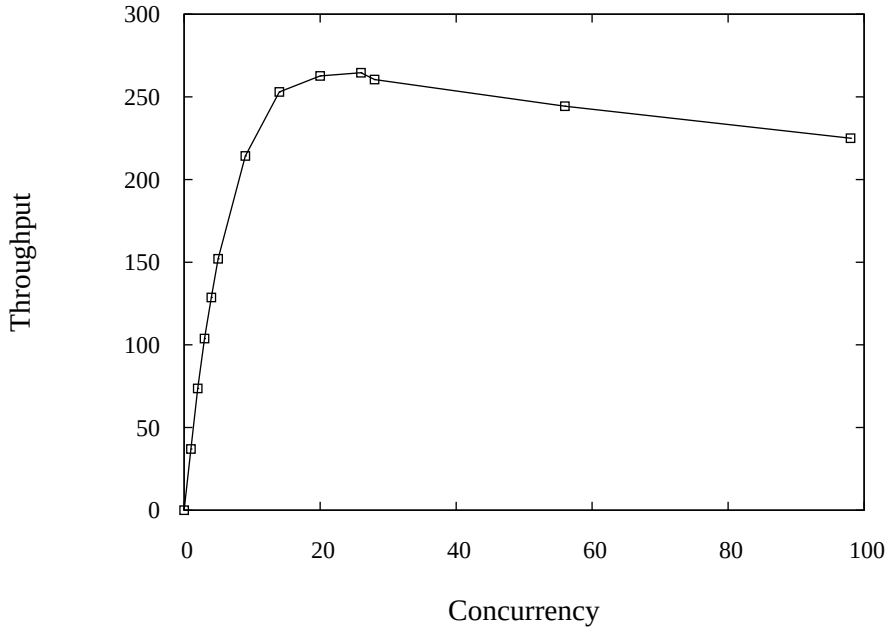

Figure 6.21: The Throughput of the System with all 14 Node as QueryNodes.

6.3 Scalability

In this section, we run multi-query experiments to evaluate the scalability of the system which contain both blob data and nonblob. In these experiments, the nonblob data are Flickr datasets, F1 and F2, which are described in Table 6.1 and the number of Querynodes is set to 4. The following three queries are run:

1: Given a user name, retrieve all photos of his /hers.

2: Retrieve 20 photo which are tagged with ”sea”.

3: Given a photo id, retrieve the photo.

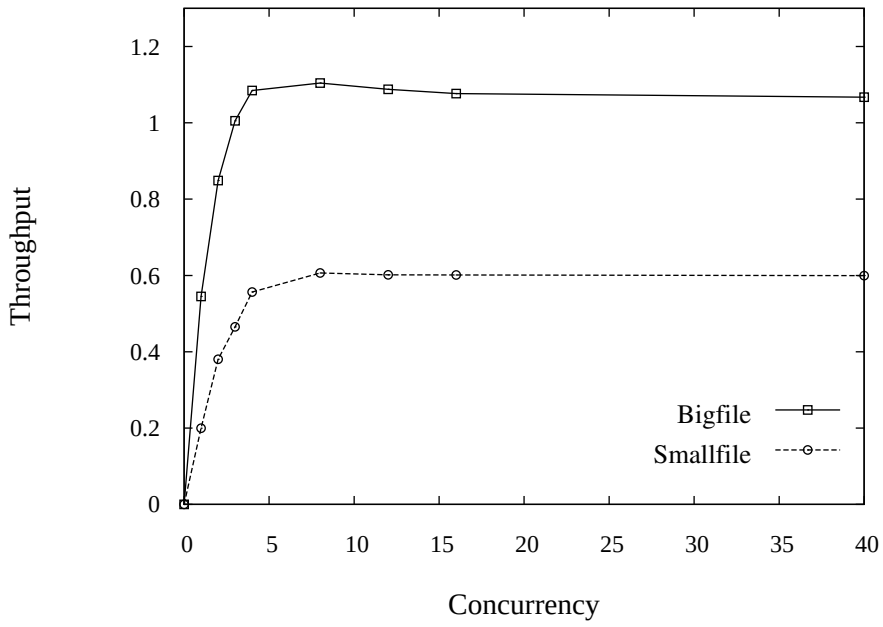

Figure 6.22: The Throughput on F1.

We run our concurrency experiments on F1 and F2 to compare the Bigfile method with the Smallfile method. The results indicate that the Bigfile method outperforms the Smallfile method as shown in Figure 6.22 and Figure 6.23.

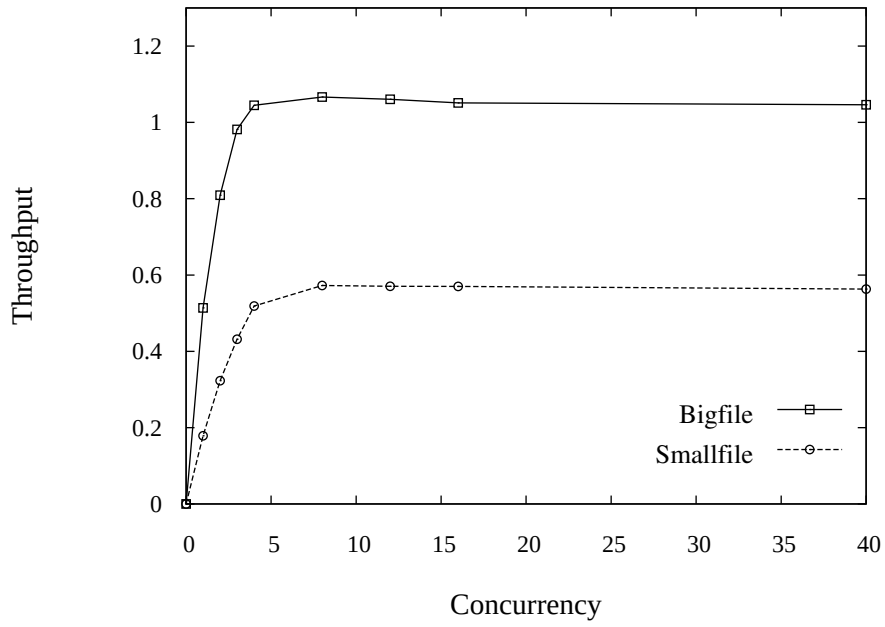

Figure 6.23: The Throughput on F2.

Chapter 7

Conclusions

The popularity of social network services and the limitations of existing systems to support such services have driven to develop a new type of systems to support social network services. In this thesis, we introduce a new data storage to store both nonblob data and blob data for social network services. We store nonblob data in our graph storage system. In the graph database system, we store each node as an object and each edge as a tuple in a table. Typically, social network services serve a large number of users and one server cannot handle all of request form them. Therefore, we also provide to two approaches, the Ordering partition method and the MST partition method, to partition a huge social network graph into several small parts. Indexing is the most important and fastest approach which reduces high I/O cost effectively and greatly improves the speed of data retrieval. We investigate two types of indexes: content index and node index. For blob data storage, we investigated an object store, HadoopObS, to manage blob data for social network services. HadoopObS is an object store which is designed to manage a massive number of photos for read-intensive applications in social network services.

Finally, we conduct some experiments to evaluate our data storage. For nonblob data, we conduct experiments based on two types of partition method, the Ordering method and the MST method. The results show that our methods outperform the conventional method on both performance and scalability. We also measure the read and write performance of our HadoopObS compared with the traditional file system to evaluate our blob data storage design. The read throughput of HadoopObS is three times of the read throughput of the traditional file system.

7.1 Future Work

In this thesis, we propose a data storage design and two partition methods for our graph database to manage nonblob data for social network services. We also introduce two types of indexes, content index and node index, to improve the query performance. We simulate our graph database on a relational database to do evaluation. For blob data storage, we design an object store on Hadoop Distributed File System, called HadoopObS, to store blob data. HadoopObS overcomes some limitations of existing systems. In the future, our graph model and storage system should be implemented in a graph database system which is designed to support social network services. Other components of this graph database system also should be investigated and implemented, such as query language, query optimizer, query processor and etc. Finally, the graph database should be combined with HadoopObS to provide data storage for both blob and nonblob data of social network services.

Bibliography

[1] Compete. http://www.compete.com.

[2] Flickr. http://www.flcikr.com.

[3] Hadoop. http://hadoop.apache.org/.

[4] Haystack. http://www.facebook.com/note.php?note_id=76191543919.

[5] Hbase. http://hadoop.apache.org/hbase/.

[6] D. J. Abadi. Column Stores for Wide and Sparse Data. In *CIDR*, pages 292–297, 2007.

[7] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML Schema to Relations: A Cost-Based Approach to XML Storage. In *ICDE*, pages 64–, 2002.

[8] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD Conference*, pages 479–490, 2006.

[9] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[10] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML). *World Wide Web Journal*, 2(4):27–66, 1997.

[11] D. Chamberlin. XQuery: a query language for XML. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 682–682, New York, NY, USA, 2003. ACM.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006.

[13] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *VLDB*, pages 263–274, 2002.

[14] K. W. Chong, Y. Han, and T. W. Lam. Concurrent threads and optimal parallel minimum spanning trees algorithm. *J. ACM*, 48(2):297–323, 2001.

[15] C.-W. Chung, J.-K. Min, and K. Shim. APEX: An Adaptive Path Index for XML Data. In *SIGMOD Conference*, pages 121–132, 2002.

[16] J. Clark and S. Derose. XML Path Language (XPath) Version 1.0. Recommendation http://www.w3.org/TR/1999/REC-xpath-19991116, World Wide Web Consortium, November 1999.

[17] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[18] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *VLDB*, pages 341–350, 2001.

[19] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[20] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, pages 29–43, 2003.

[21] R. Goldman, S. Chawathe, A. Crespo, and J. McHugh. A Standard Textual Interchange Format for the Object Exchange Model (OEM). Technical Report 1996-5, Stanford InfoLab, 1996.

[22] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *WebDB (Informal Proceedings)*, pages 25–30, 1999.

[23] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[24] A. Halverson, J. L. Beckmann, J. F. Naughton, and D. J. Dewitt. A Comparison of C-store and Row-store in a Common Framework. Technical report, 1566.

[25] A. Halverson, V. Josifovski, G. M. Lohman, H. Pirahesh, and M. Mörschel. ROX: Relational Over XML. In *VLDB*, pages 264–275, 2004.

[26] A. Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, 2009.

[27] K. Kamvar, S. Sepandar, K. Klein, D. Dan, M. Manning, and C. Christopher. Spectral Learning. Technical Report 2003-25, Stanford InfoLab, April 2003.

[28] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD Conference*, pages 779–790, 2004.

[29] W. Kim. Research Directions in Object-Oriented Database Systems. In *PODS*, pages 1–15, 1990.

[30] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, February 1956.

[31] B. Li, M. Hui, J. Li, and H. Gao. iVA-File: Efficiently Indexing Sparse Wide Tables in Community Systems. In *ICDE*, pages 210–221, 2009.

[32] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *VLDB*, pages 361–370, 2001.

[33] C. Lin. Object-Oriented Database Systems: A Survey. 2003.

[34] S. Lloyd. Least squares quantization in PCM. *Information Theory, IEEE Transactions on*, 28(2):129–137, Mar. 1982.

[35] S. Mitra, A. Bagchi, and A. K. Bandyopadhyay. Design of a Data Model for Social Network Applications. *J. Database Manag.*, 18(4):51–79, 2007.

[36] B. C. Ooi, B. Yu, and G. Li. One table stores all: Enabling painless free-and-easy data publishing and sharing. In *CIDR*, pages 142–153, 2007.

[37] Y. Papakonstantinou, H. Garcia-molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *In Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, 1995.

[38] R. Sears, C. van Ingen, and J. Gray. To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem? *CoRR*, abs/cs/0701168, 2007.

[39] J. Shanmugasundaram, E. J. Shekita, J. Kiernan, R. Krishnamurthy, S. Viglas, J. F. Naughton, and I. Tatarinov. A General Techniques for Querying XML Documents using a Relational Database System. *SIGMOD Record*, 30(3):20–26, 2001.

[40] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. Dewitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. pages 302–314, 1999.

[41] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[42] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD Conference*, pages 204–215, 2002.

[43] P. Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.

[44] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *SIGMOD Conference*, pages 110–121, 2003.

[45] M. Zand, V. Collins, and D. Caviness. A Survey of Current Object-Oriented Databases. *DATA BASE*, 26(1):14–29, 1995.