Database Indexing for Skyline Computation, Hierarchical Relational Database, and Spatially-Aware SPARQL Evaluation Engine

by

Chih Jye Wang

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
December 12, 2015

Keywords: Database management systems, R-Tree index, skyline computation, index-based pruning skyline, hierarchical relational database, SPARQL, spatial SPARQL, fusion experiments, control systems, MDPX

Approved by

Wei-Shinn Ku, Chair, Associate Professor of Computer Science and Software Engineering
Xiao Qin, Professor Computer Science and Software Engineering
James Cross, Professor of Computer Science and Software Engineering
Alvin Lim, Professor of Computer Science and Software Engineering
Edward Thomas, Jr., Professor Physics

Abstract

Computerized databases are ubiquitous and play an important role in many software projects. The efficiency of these systems are crucial in applications they support. Data indexing is the key to efficient data retrieval in database systems. This dissertation considers three data application domains and operations in which data indexing can greatly improve the turnaround time of request operations.

The first part is the study of multi-dimensional and multi-criteria skyline computation. Skyline is a subset of data records that are the best in a data set, where best is defined by user criteria and preference. In our technique, we first create an index using the R-Tree structure. We then propose a greedy algorithm that utilizes the R-Tree index to aggressively prune the search space as the index is traversed. Experimental evaluation shows that our algorithm performs better than previous algorithms.

In the second part of this dissertation, we consider the efficiency of querying hierarchical data in a relational database (hierarchical relational database). The relational database model is known to efficiently store and retrieve data stored in flat tables; however, it was not designed for hierarchical data. Due to this limitation, many operations – such as answering membership on a hierarchical data – require recursion or iterations. In this study, we propose an index structure based on the nested-sets model. With this index, we can eliminate recursions and allow database engines to perform hierarchical operations with index scans.

In the last part of this dissertation, we consider storage and retrieval of point-based spatial data (POIs) in the Geo-Store RDF Triple Store. When we first started this study, spatial data management in RDF was still a fairly new research area and there were no standards for query spatial data using SPARQL. In this work, we designed an index scheme using the Hilbert Space-Filing Curve (HSFC). Leveraging the spatial locality of HSFC, we are able to efficiently evaluate

spatial queries such as range and k-nearest-range (kNN) queries. In this work, we define range and kNN SPARQL filters. We implemented these filters and showed that our implementation handles spatial queries better than other triple stores.

Acknowledgments

I would first like to thank my advisory committee for the guidance and support. I want to thank my advisor, Dr. Wei-Shinn Ku, for his guidance and mentoring throughout my graduate studies. His expertise in the fields of spatial database management has facilitated my research and the completion of my dissertation. This dissertation would not be possible without his insight.

I want to express my gratitude to my advisory committee member, Dr. James Cross, whose careful review of this dissertation and expertise of technical writing made this dissertation as good as it can be. His character of excellence has propelled me to strive for excellence in my work.

I would like to thank my advisory committee member, Dr. Xiao Qin, for helping me to revise this dissertation. I am especially thankful for his suggestions in correcting logical and grammatical errors in early drafts. In addition, I also want to thank him for career advice.

I am thankful for the advice and words of encouragement of my advisory committee member, Dr. Alvin Lim. His knowledge in semantic web and related works has offered a unique perspective to ensure the quality of this dissertation.

I want to especially thank Dr. Edward Thomas, Jr. for serving as the University Reader of my advisory committee. I am grateful for the discussions we had during my graduate study and his support for me to finish my degree.

I want to give special thanks to my parents and my family for the unwavering support to help me pursue my degree. I would not be able to complete this dissertation without their support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

As our society moves toward digitization of the world around us and increasing reliance on software applications, computerized database systems have became prevalent and important. As the number of software applications and services (such as Facebook and Gmail) grow, so do the amount of data we generate. Database management systems (DBMS) are the core of most software applications; therefore, the efficiency of the DBMS are crucial to the applications they support. Here we distinguish software applications and DBMS as follows: applications are the software services that consist of the logic of the software application and how the data are consumed; on the other hand, a database or DBMS has very little logic. The purpose of DBMS is to store and efficiently retrieve data and that is it. A database may be able to support multiple applications.

The earliest form of databases are physical written records on paper or other forms. Computerized databases are also not a new concept. Since the earliest computer systems, software applications have always had the need to save and store data; therefore, a computer file is essentially a database if the file can be consistently read and stored. The earliest DBMS utilized the hierarchical data model [99] that was first described in the 1960s and implemented by the IBM Information Management System (IMS) [41, 65]. The relational model [13] was described by Edgar Codd in 1970, and remained the most prevalent database model in the software industry. The relational model is the basis for popular database management systems such as MySQL, Microsoft SQL Server, Oracle Database, and SQLite.

Despite the fact that database management systems have been commonplace in software projects, there are rooms for improvement in specific application domains. The growth of volume and diversity of data on the Web has also ushered in challenges that motivated new flexible data models, such as the Resource Description Framework (RDF) for semantic web data. This

dissertation addresses three application domains and operations in which through data indexing techniques, the query efficiency can be improved. The addressed application domains are skyline computation, hierarchical data in relational databases, and point-based spatial data (POIs) in RDF triple stores.

This dissertation is organized as follows. The rest of this chapter provides an introduction to the chapters covered in this dissertation. Starting from Chapter 2, each application domain is discussed in detail, where each chapter is structured as a standalone discussion with an introduction, related works, and description of the technique. Chapter 2 covers Skyline computation in broadcast environments; Chapter 3 discusses an index technique for hierarchical data in relational databases; Chapter 4 describes Geo-Store: a spatially-aware RDF triple store and SPARQL query engine; Chapter 5 concludes this dissertation and provides future work; and Appendix A provides a case study of computerized control systems for scientific and industrial scale experiments and describes the current state of the control system of the MDPX experiment. This appendix also shows that many of the topics covered in this dissertation may be used in practice.

1.1 Skyline Evaluation in Wireless Data Broadcast Environments

Given a data set T, the skyline is an operation that finds a subset S of T that is the best or the most efficient in T, where 'the most efficient' is defined by the user of the query. The proper definition for the skyline operator is that for a record, s, to be in S, the record s must not be dominated by other records in the original data set T [54, 74]. For example, someone who is planning for an ocean-view vacation would be interested in finding a list of hotels that are close to the ocean and at the same time not too expensive. In this case, the user preference for our query is to minimize both the distance to the beach and the cost of the hotel. If a hotel a is the same distance to the beach as b, but has higher cost, then a is dominated by b; therefore, a cannot be in S.

Skyline computation in broadcast environment is described in Chapter 2, which is divided into two parts. The first part describes the constraints in data broadcast environments and proposes

tree-based distributed index (TDI), a data serialization technique to address these constraints. The second part of the chapter describes our skyline computation technique, index-based pruning skyline (IPS). In IPS, we construct a data index of the data set T using R-Tree spatial index. After that, IPS computes skylines by aggressively pruning the search space as the index is traversed by the search algorithm. Because IPS prunes the search space earlier than record-based pruning techniques, IPS performs better than record-based pruning algorithm.

1.2 Hierarchical Data in Relational Database Management Systems

Relational database are one of the most prevalent categories of database management systems used in the software industry due to its performance and the standardized query language SQL. However, the relational model is not optimized for storing data that models hierarchical relationships. Microsoft has recently announced the support of hierarchical data in Transact-SQL starting in SQL Server 2008 [64]; however, there is limited to no support in other systems.

The simplest and most intuitive way to represent hierarchy in relational databases is using the adjacency list model. In this model, every record in the hierarchical relation (table) has a pointer to the parent record in the same relation. One drawback of the adjacency list model is that some hierarchical queries and operations require recursion or iteration. Having to perform iterations means that our application has to submit multiple queries to the database server to accomplish the operation.

In Chapter 3, we describe another model to store hierarchy in relational databases based on nested sets structure and is an extension to the adjacency list model. In our approach, we add two additional fields, left and right, which are augmented to each record to denote the hierarchical relationships. The two fields are then indexed in B-Tree to allow querying relationship using only index scans. In the chapter we first describe the hierarchical operations we use in the chapter; then, we describe the storage and index technique. Our design performs much better than the adjacency list model because our technique can eliminate recursions in many cases.

1.3 Geo-Store: A Spatially-Aware SPARQL Evaluation Engine

As the data of an application becomes more diverse or originates from multiple sources, a more schema-relaxed data model is desired to simplify the process of data integration. The Resource Description Framework (RDF) is a schema-relaxed data storage model that is used to store the diverse data of the Web. RDF stores data as a set of statements each consisting of three parts: subject, predicate, and object. The subject is the entity being described; predicate is the property of the subject; and object is the value of the property. For this reason, RDF databases are often called triple stores. Using this format, an RDF database can have loose schema or be completely schema free.

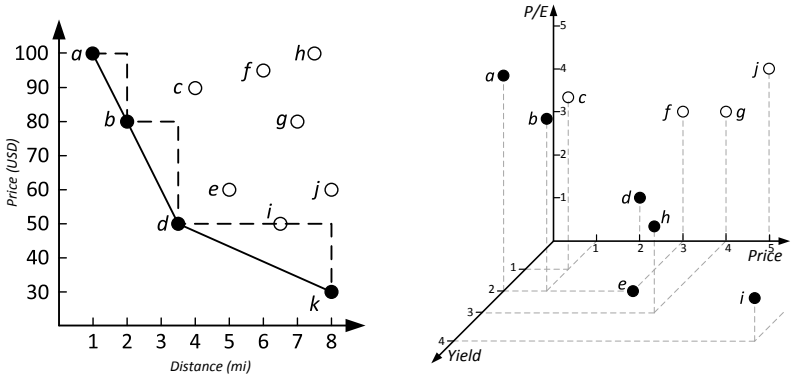
In Chapter 4 of this dissertation, we specifically focus on point-based spatial data (POIs) in RDF triple stores. Our contribution in this chapter is three-fold. The first is to define range and k-nearest-neighbor query filter in SPARQL. The second is a unique spatial data index technique utilizing the Hilbert Space Filling Curve (HSFC). The third is an implementation of our query filters and index techniques in our Geo-Store triple store.

Chapter 2

Efficient Evaluation of Skyline Queries in Wireless Data Broadcast Environments

2.1 Introduction

Skyline is a query type that retrieves data items that are considered "interesting objects" with respect to multiple attributes of the data set. For example, someone who is planning for an ocean-view vacation would be interested to find a list of hotels that are close to the ocean and at the same time not too expensive. A hypothetical data set of hotels is shown in Table 2.1(a). The two relevant attributes in this case are *minimum distance* to the ocean and *minimum price*. Figure 2.1(a) shows the skyline points in solid dots as the operation finds hotel records that are successively further from the ocean, but the prices are the best for such distances. In the figure, hotel a is a skyline point because it is the closest to the ocean, although it is not the lowest price. Hotels b and d are part of the skyline because each have the closest distance at its price. Hotel k is a skyline point because it is the cheapest of all the hotels.



(a) 2-D skyline of hotels of (X = min, Y = min)

(b) 3-D skyline of stocks with attributes (X = min, Y = min, Z = max)

Figure 2.1: Sample skylines.

Skyline has broad applications and relevance to many fields. Skyline computation in real-time streaming systems with frequent data updates has been studied in [63] and [88]. For distributed web

Table 2.1: Sample data sets.

(a) 2-D data of Fig 1(a)

Hotel	Dist.	Price
a	1	8
b	2	6
c	4	7
d	3.5	3
e	5	4
f	6	7.5
g	7	6
h	7.5	8
i	6.5	3
j	8	4

(b) 3-D data of Fig 1(b)

Co.	Price	P/E	Yield
a	0	5	2
b	1	4	2
c	1	4	1
d	2	2	0
e	3	0	2
f	3	3	0
g	4	3	0
h	4	2	3
i	7	1	4
j	5	4	0

services, skyline query solutions have been proposed in [2]. Skyline query has also been applied in sensor networks in [11]. Meanwhile, the data broadcast model is a scalable way to disseminate data (e.g., FM broadcasting). Unlike the on-demand model, such as most of the services on the Internet, the broadcast model can scale almost infinitely. However, a challenge and drawback of this model is the forward-only access characteristic. While many studies have been done on different query types to support efficient query evaluation in broadcast environments [31, 34, 57, 58, 104], to the best of our knowledge, the work in [31] is the only study on skyline query processing in broadcast environments. Although [31] considers broadcast efficiency (more details in Section 2.2.2), the proposed solution is unable to support all possible skyline types (i.e., combinations of min and max attributes). In addition, the work did not address details for skyline computation of higher data dimensions (> 2 dimensions) as illustrated in Figure 2.1(b), which plots the potential investment targets (solid dots) based on the stock information in Table 2.1(b). For each stock listed, we want the price and price-to-earnings ratio to be minimized and the yield to be maximized. To address these challenges, we design a flexible broadcast index and a skyline evaluation algorithm that utilizes the index to efficiently evaluate skyline queries in broadcast environments. Specifically, the contributions of this research are as follows:

- We design a flexible on-air index that supports skyline query evaluation in data broadcast environments for an arbitrary number of data dimensions.

- We propose an efficient data broadcast skyline query algorithm which supports users to ask for minimal or maximal records in each dimension.
- We evaluate the performance of the proposed algorithms through extensive experiments.

The rest of this chapter is organized as follows. Section 2.2 provides background knowledge of the skyline operator and wireless data broadcast. Section 2.3 surveys related works. We introduce our index structures to facilitate broadcast skyline query in Section 2.4. In Section 2.5 we propose our pruning region based technique for skyline query evaluation. The experimental validation of our design is presented in Section 2.6. Conclusion and future work is described in chapter 5.

2.2 Preliminary

2.2.1 Skyline Query

Skyline query is an operation that finds all data records that are not dominated by any other data records in a given data set. A record dominates another record if it is as good or better in all dimensions and better in at least one dimension. Skyline query is closely related to the maximal vector problem [26, 59]. Here, we briefly introduce these concepts.

Definition 2.2.1 (Maximal Vector)**.** *Given the set of vectors V, for $v \in V$, if there does not exist $u \in V$ such that $u \geq v$ and $u \neq v$, then v is a maximal vector of V.*

$$MaxVect(V) = \{v | \forall v, u \in V \nexists u \geq v\} \tag{2.1}$$

The comparison operator $\geq$ utilized in Equation 2.1 is defined on vectors such that for two vectors u, $v \in V$, $u \geq v$ if $u_i \geq v_i$ for i = 1 to n and vice versa for the $\leq$ operator. In other words, a vector v is greater than the other vector u if and only if all elements of v is *greater than or equal to* all elements of u. The comparison operators for each element of the vectors are the natural ordering operators defined on each domain.

The main difference between the maximal vector operation defined in Definition 2.2.1 and the skyline operation is that the skyline operator defines a set of *preference specifiers*, σ, which are to be either *minimal* (*min*) or *maximal* (*max*) for each dimension.

Definition 2.2.2 (Skyline Operator)**.** *Give a set of tuples, T, and a set of preference specifiers, σ, the skyline operator is defined as follows:*

$$Skyline(T, \sigma) = \{t \in T | \nexists s \in T \text{ where } s \succ t\} \tag{2.2}$$

The dominance relationship $\succ$ in Definition 2.2.2 is defined in the following definition.

Definition 2.2.3 (Dominance Relationship)**.** *Given two tuples t_1 and t_2 in T, t_1 dominates t_2 if and only if all attributes of t_1 dominate or equal all attributes of t_2 and at least one attribute of t_1 dominates. Dominance for each attribute depends on the preference specifier, σ_i, for that attribute. Given x and y from a certain domain D_i, the dominance for the attributes is defined as follows:*

1. *If σ_i =* min*, x dominates y if $x < y$.*
2. *If σ_i =* max*, x dominates y if $x > y$.*

2.2.2 Wireless Data Broadcast

A wireless data broadcast environment consists of a broadcast channel, a broadcast server, and mobile clients who are interested in the broadcast program from the server. The server is the originator of the broadcast program which contains relevant data records and pushes the data onto the channel. The mobile clients receive desired data by listening to the channel. Examples of services such as XM Satellite Radio[1] and Microsoft MSN Direct. The model is illustrated in Figure 2.2.

A complete dissemination of all the records in the data set from the server is a *broadcast cycle*. A cycle follows another cycle (see Figure 2.3). In this chapter, we use the terms cycle and program interchangeably to denote the content of the broadcast channel.

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

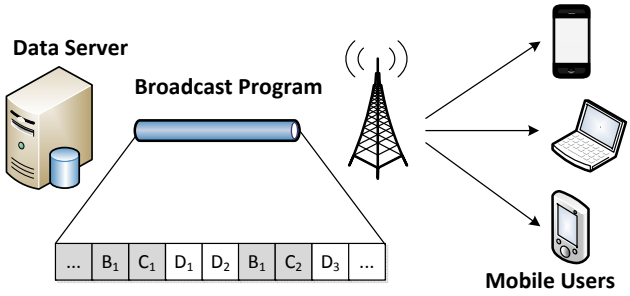


Figure 2.2: A wireless broadcast environment.

An inherent challenge of the broadcast model is the forward only data model; there is no random access of the data set. When a client misses a piece of data from the current cycle, the client must wait until the next cycle. The influences of these properties on the broadcast environment are: (1) we cannot adopt skyline algorithms designed for disk-based database systems that require random data access, and (2) we must design a self-explanatory broadcast program using indexes.

Many previous studies have been done on different broadcast environments. Multi-channel broadcast for data dissemination (and techniques of data allocation) has been studied in [33, 40, 103]. Adaptive broadcast systems that allow limited uplink (client to server) communication have been surveyed in [102]. In this chapter, we assume the following properties for our broadcast environment: (1) a channel is forward-only, (2) only one channel is utilized, and (3) no uplink from clients to server.

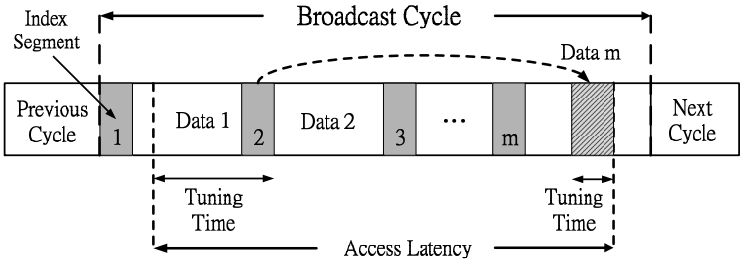
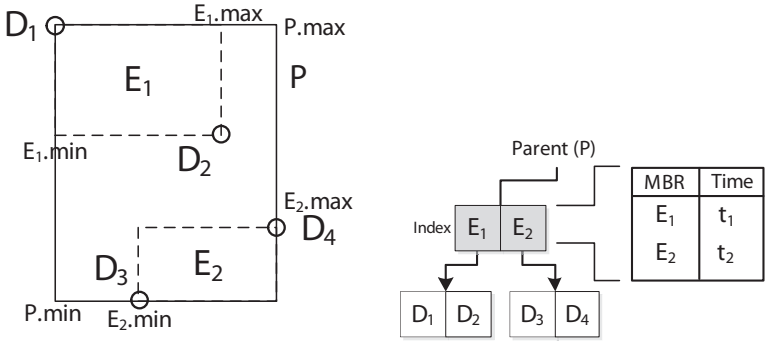


Figure 2.3: Broadcast program cycles.

Power consumption is a major concern in wireless broadcast environments since most clients are mobile devices, such as cellular phones, powered by a small battery. Therefore, power is a scarce and valuable resource for these devices; continuously listening to the broadcast channel is expensive in terms of power usage. Most mobile devices are able to turn off the radio receiver

when they are not actively receiving data to conserve power. To reduce power consumption, an index is used to make the broadcast cycle self-descriptive. As depicted in Figure 2.4, an index provides additional information in a broadcast program to tell the clients the approximate time that a data record will be broadcast. A broadcast index is analogous to an index in traditional databases in which the indexes provide the location of data records and facilitate fast lookup of records. With the index information, the clients can turn off the radio receiver to conserve power and only tune into the channel when the desired data is being broadcast.



(a) Data set. (b) On air index.

Figure 2.4: A sample broadcast index.

Adding an index to a broadcast cycle also adds space overhead to the cycle and consumes broadcast bandwidth. Quantities that measure the efficiency of a wireless broadcast program are defined below. *Access latency* and *tuning time* are well-known metrics of the broadcast model and have been studied in [44, 57, 58, 62, 104]. We also define *index percentage* and *initial index probe* as measurements of index space overhead and index probe latency.

Definition 2.2.4 (Access latency)**.** *The amount of time from when a client requests data to the point when the client receives all the desired data. This quantity is denoted by* α*.*

Definition 2.2.5 (Tuning time)**.** *The total amount of time a client actively listens to the channel. The time determines the power consumed by the client to retrieve the required data. The quantity is denoted by* τ*.*

Both the access latency and the tuning time are measured in terms of number of data packets.

Definition 2.2.6 (Index percentage)**.** *The ratio of the space allocated to the index to the space of the entire cycle measures in bytes. This quantity is defined as* $\rho = \frac{\delta}{\omega}$.

Definition 2.2.7 (Initial index probe)**.** *The amount of time for a client to get to the first index segment. This quantity is denoted by* λ.

Table 2.2: Summary of notations.

Notation	Description
n	Number of dimensions of skyline
m	Number of index segments in a cycle
D	A set of attribute domains D = $\{d_1$, ..., $d_n\}$
T	A set of records (tuples)
σ	A set of skyline preference specifiers
B	A minimal bounding box (MBR)
E	An R-tree index entry (MBR, time)
b	Branching factor of an index tree
h	Height of an index tree
L_r	Levels of index tree replication
ρ	Index percentage
α	Access latency
τ	Tuning time
δ	Size of a broadcast index
θ	Size of a broadcast data set
ω	Size of an entire program cycle $\delta + \theta$

2.3 Related Work

2.3.1 Skyline Computation

Algorithms of the computation of skyline records in traditional database systems have been studied in [6, 54, 74]. The well-known algorithms are Nested-Loop, Block-Nested-Loop, and Divide and Conquer have studied in [6]. The Nested-Loop algorithm is a naive algorithm that it compare every record with every other record. In Block-Nested-Loop, a record is only compared with other records in the same block. In Divide and Conquer (D&C) utilizes divide and merge strategy to computer skyline.

Branch-and-Bound Nearest Neighbor [74] algorithm utilizes a tree index to efficiently computes skyline. In the design, data records are represented geometrically and a tree index is built on the data. The algorithm then uses branch-and-bound tree traversal to find nearest neighbors as skyline. Our algorithms uses a geometric tree index, but BBS-NN cannot be easily adopted to the broadcast environment due to that the algorithm requires backtracking of the tree.

Extension SQL syntax for skyline was defined by Börzsönyi et al. in [6]. The syntax defines an additional SKYLINE clause in SQL that specifies how the skyline operation should be performed.

2.3.2 Wireless Broadcast Index

The intuitive technique of no index and one-time index at the beginning of a broadcast cycle has been considered in [44]. With no index, the length of a cycle is minimized, but the tuning time is the entire cycle since the program is not self-descriptive. With one-time index, the clients are able to filter unwanted data and reduce tuning time, but if a client misses the one-time index, then it will have to wait until the next cycle even if there is useful data in current cycle.

$(1, m)$ indexing, proposed by Imielinski, *et al.* in [44], is a mitigation to the problem of one-time index by replicating the entire index every $1/m$ length of the broadcast cycle. The benefit of this index technique is that when a client misses an index segment, it can wait for the next index in the same cycle [58]. The drawback of this technique is the space consumption of replicating the full index several times in the cycle.

Distributed index was also proposed in [44]. This index also replicates the index in the broadcast cycle, but only a part of the index is replicated. Advantages of this method are (1) an index can be obtained throughout a cycle, (2) reduced bandwidth consumption compared with $(1, m)$, and (3) the method is not limited to any particular index structure.

Distributed index for spatial data in error-prone air broadcast was introduced in [104]. Instead of replication, this paper proposes a distributed index in the broadcast cycle with no duplicate indexes. The proposed solution indexes broadcast spatial data using Hilbert values. Each data

record contains an index table that includes the data segments that will be pushed onto the channel in the near future.

2.4 Tree-based Distributed Index

In this section we introduce how the index and data are allocated on the broadcast channel to form the broadcast program. We introduce a Tree-based Distributed Index (TDI), our index allocation technique, which is based on the distributed index allocation proposed in [44]. The benefits of a distributed index over other allocation methods will be discussed in Section 2.3.2.

We utilize an R-tree [30] to index our multi-dimensional data records. We assume that each node of an R-tree consists of b number of entries, $\{E_1, E_2, ...E_b\}$, where b is the branching factor, or the number of children at each internal node, of the index tree, as illustrated by Figure 2.4. Each entry contains a pointer to a child index node (if the node is not a leaf node), or a pointer to a set of data records (if the node is a leaf node). The pointer contains the time when the child item will appear on the broadcast channel. An advantage of utilizing the R-tree is that the indexed broadcast data can be employed to support other spatial query types, such as range [73] and kNN queries [37].

2.4.1 Broadcast Structure

A broadcast program cycle is a linear representation of the index tree and data and consists of *index segments* and *data segments*. Index segments contain temporal pointers to either another index segment or a data segment. Data segments contain actual data records. Index and data segments are interleaved to form a broadcast program. Each index segment is further divided into smaller units called buckets. Buckets are logical independent units that represent a portion of the tree index. The purpose for buckets is that a client does not have to download an entire index segment if it only needs a bucket.

In addition to a list of temporal pointers, each index bucket also contains a pointer to the next index segment and a pointer to the beginning of next broadcast cycle. The purpose of these

pointers is to direct the client to the next index segment in the case that the client tunes in at the index bucket but is not interested in the data pointed by the index.

To save bandwidth, data segments do not contain any index information other than data records. The broadcast program structure is illustrated in Figure 2.5.

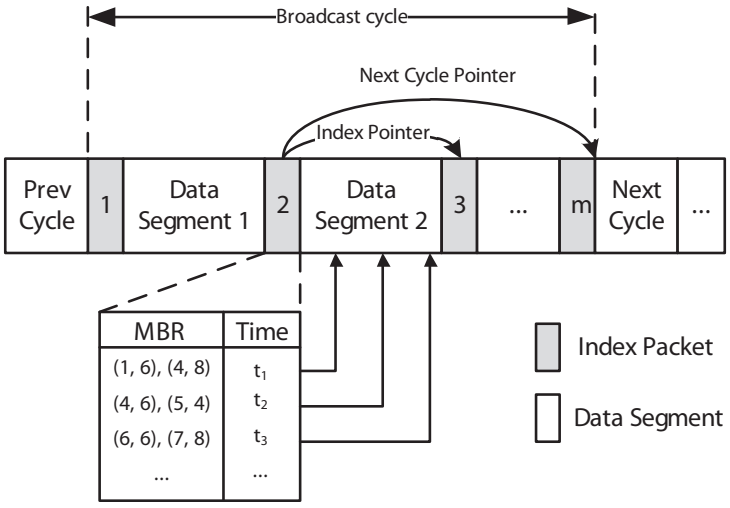


Figure 2.5: Broadcast program cycle format with index segments and data segments.

2.4.2 TDI Allocation

After indexing the data set with an R-tree, the next step is to publish both data and index onto the linear broadcast channel. TDI allocates space for index and data by performing a depth-first traversal of the index tree as illustrated in Figure 2.6. In this process, the root index node A is first included in the cycle because it is first traversed. Index node B_1 is then included in the program followed by C_1, then the data items D_1 and D_2. We utilize depth-first traversal of the index tree instead of breadth-first traversal in our design because breadth-first traversal cannot distribute the appropriate portion of index that exactly indexes the data segment which follows it.

TDI replicates the top L_r levels of the index tree b times, where the root node is considered level 0. Therefore, the replication is not an entire path replication of the index. The replication helps the clients get a broader picture of upcoming broadcast items. Given n_i to be the current index node to be broadcast, if the level of n_i is L_r or less, then the parent of n_i is replicated.

Otherwise the parent is not replicated. An example is shown in Figure 2.6; the root and the second level index nodes are replicated. Of course, the best view of upcoming data records would be to replicate the entire index, but this would be a complete replication, which we try to avoid to decrease broadcast cycle length.

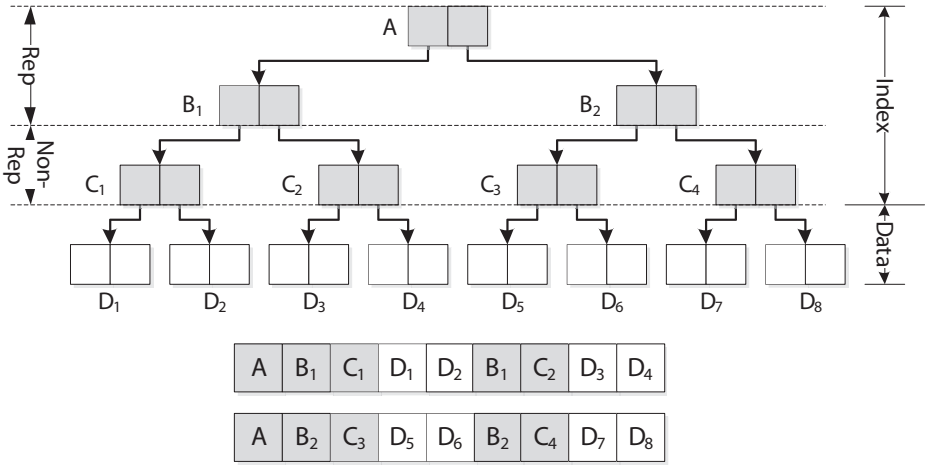


Figure 2.6: Index Structure.

2.4.3 Analysis

This section presents the derivation of the two performance evaluation metrics of the tree-based distributed index, which we defined in Subsection 2.2.2.

Index Percentage

Index percentage measures the space overhead of the index structure. It is defined as the ratio between the space allocated to the index on the broadcast cycle to the length of the entire broadcast cycle.

Let L_r be the levels of replication (for example, L_r for the TDI in Figure 2.6 is 2) and η be the space required for an index node. The number of nodes replicated in the cycle is $(\sum_{i=0}^{L_r-1} b^i)$. In TDI, each replicated index node appears b times in one cycle. Consequently, the total space taken by the index is the space for all index nodes plus the additional space for replication. It is given by:

$$\delta = \eta\{(\sum_{i=1}^{h-1} b^i + 1) + (\sum_{i=0}^{L_r-1} b^i)(b-1)\} \tag{2.3}$$

The length of the broadcast cycle ω is the space of the index plus the space of the data ($\delta + \theta$). Therefore, the index percentage is defined as:

$$\rho = \frac{\delta}{\omega} \tag{2.4}$$

Initial Index Probe

The tree-based distributed index divides the entire data set into smaller data segments and reduces the initial probe of the first index segment. The length of each data segment, denoted by ℓ, is determined by the number of index segments distributed among the broadcast cycle. As one can see in Figure 2.6, there are four leaf nodes in the index and the data is divided into four segments. Therefore, the number of data segments is b^{L_r} and the size per data segment is:

$$\ell = \frac{\theta}{b^{L_r}} \tag{2.5}$$

The initial index probe is half of the length of an index segment plus a data segment:

$$\lambda = \frac{1}{2}(\frac{\delta}{b^{L_r}} + \ell) \tag{2.6}$$

2.5 Pruning Region Broadcast Skyline

In this section, we present skyline computation algorithms. We first describe the concept of a pruning region, on which our algorithms are built. After foundations, two skyline algorithms are presented: Record-Based Pruning Skyline (RPS) is presented in Section 2.5.1 and Index-Based Pruning Skyline (IPS) is presented in Section 2.5.2. Both RSP and IPS utilize the tree-based distributed index described in the previous section to build a set of pruning regions to eliminate

unwanted data records. IPS generally provides better performance, but only produces correct result when the bounding boxes of the index tree does not overlap; whereas RSP works for any tree index.

The rest of this section describes pruning region. In the following definitions, when we say a point, q, and region, R, we mean a point and region in the data space such that (q_1, ..., q_n) are record attributes.

Definition 2.5.1 (Pruning Region)**.** *A pruning region, $R(d, \sigma)$, is a rectangular region of data space that is dominated by one or more data records. d is a point called dominant point and σ is an ordered list of preference specifiers.*

Given an arbitrary point d and σ, a pruning region R is defined such that for each attribute i, if $\sigma_i = min$, then d_i is the lower bound of R for the ith dimension and the region extends to the maximal value for the ith dimension. Similarly, if $\sigma_i = max$, then d_i is the upper bound and extends to minimal value of the dimension.

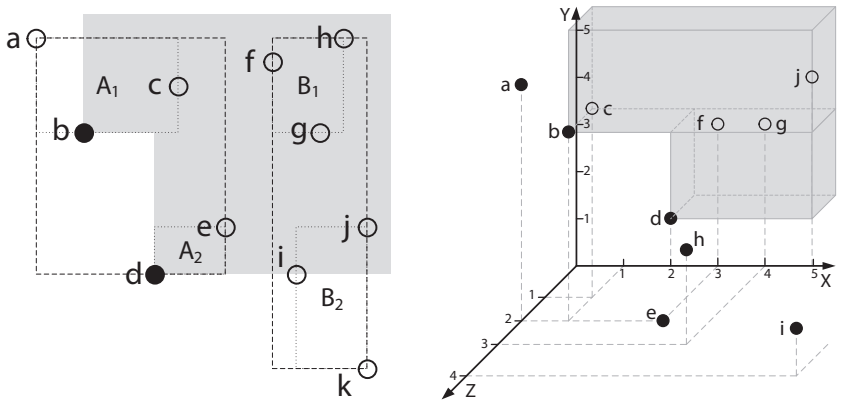
A dominant point can also be defined with any rectangle as follows:

Definition 2.5.2 (Dominant Point)**.** *Given a n-dimensional rectangle, B, and a list of skyline preference specifiers, σ, the dominance point of the rectangle $d(B, \sigma)$ is the point of intersection of all axes of the data space and contains the "best" of all axes in B, where the "best" is defined by σ.*

The ideas of dominant point and pruning region are illustrated in Figures 2.7 where pruning regions are illustrated as the gray regions and dominant points are solid dots. The figure shows a 2-dimensional (Figure 2.7(a)) and a 3-dimensional (Figure 2.7(b)) data space with 2 pruning regions defined by dominant points b and d. As illustrated, a pruning region is a region of an n-dimensional space that has been dominated by pivot point and can be ignored in the upcoming broadcast data stream. By Definition 2.5.2, we have the following properties:

Property 2.5.1. *Given σ, a pruning region can be constructed from any arbitrary data record, p, as $R(p, \sigma)$.*

Property 2.5.2. *The dominant point of pruning region R dominates all data points and minimal bounding rectangles inside R.*



(a) 2-D space with σ = (min, min). (b) 3-D space with σ = (min, min, max).

Figure 2.7: Pruning region with candidate Skyline points.

Our skyline algorithms utilize pruning regions to determine if a record (a point in data space) or a branch of the index (a bounding rectangle) from the broadcast program can be ignored. The data items (record or index) that can be ignored are the ones that are covered by (or inside) any of the pruning regions that the skyline algorithm constructs and maintains. The following lemma determines if a data record covered by a pruning region.

Lemma 2.5.1. *A data record, p, is covered by a pruning region $R(d, \sigma)$, if for each attribute i, one of the following is met:*

1. *If $\sigma_i = min$, then $p_i \geq d_i$*
2. *If $\sigma_i = max$, then $p_i \leq d_i$*

Lemma 2.5.1 says to check if a data record, p, is covered by a pruning region, all attributes of the record has to be compare against the dominant point(or record), d, of the pruning region. If any attribute does not satisfy any of the two conditions, the record is not covered.

To determine if a branch of the broadcast index is covered by a pruning region, we need to compare the minimal bounding box of the index branching to the pruning region. By Property 2.5.2, the following determines if a rectangle (bounding box) is covered.

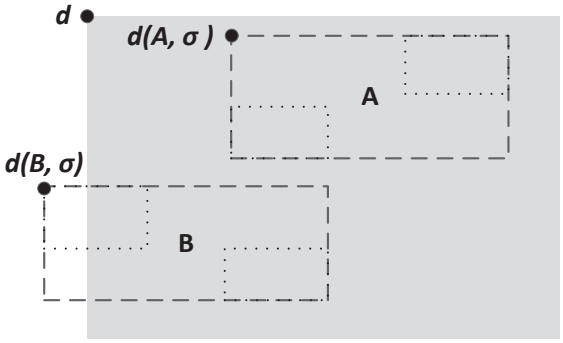


Figure 2.8: A pruning region defined by d and $\sigma = (X = min, Y = max)$.

Lemma 2.5.2. *Given a pruning region R(d, σ) and a rectangle B, R(d, σ) covers B if R(d, σ) covers the dominant point of B, d(B, σ).*

Proof. The pruning region $R(d, \sigma)$ covering the dominance point $d(B, \sigma)$ of B implies that one of the conditions of Lemma 2.5.1 is true for every attribute and that $d \succ d(B, \sigma)$. Due to Property 2.5.2, $d \succ B$. □

An example is given in Figure 2.8. A pruning region $R(s, \sigma)$ is illustrated in gray with $\sigma = (X = min, Y = max)$. Bounding box A can be pruned since its dominant point, $d(A, \sigma)$, falls inside the pruning region, while bounding box B cannot be pruned since, although it is partially covered, its dominant point, $d(B, \sigma)$, does not fall inside the pruning region.

2.5.1 Record-Based Skyline Pruning

In this section, we present our skyline algorithm that utilizes the data records received from the broadcast program to construct pruning regions. At the beginning of the algorithm, the list of pruning regions, maintained by the algorithm, is empty. At this state, the algorithm cannot prune any broadcast item (index or data); therefore stays tuned into the channel and follows the index to the first data segment. As the algorithm receives data records, it constructs one pruning region for each data record as state in Property 2.5.1. The following summarizes the notations we use in algorithm listings:

Table 2.3: Algorithmic Notations

Symbol	Description
S	A set of candidate skylines; S' denote local skylines.
R	A set of pruning regions; R' denotes local pruning region.
GetBucket(E)	Wait and download items specified by E.
Skyline(B, σ)	Compute skyline from set B via NN.
GetPRegions(S, σ)	Compute pruning regions from set S.
Prune(S, R)	Remove items from S that are covered by R.

The RPS algorithm is shown in Algorithm 1. During initialization (line 1-2) of the algorithm, the list of candidate skylines, S, and the list of pruning regions, R, are assigned the empty set.

After initialization, the algorithm goes into the outer loop (line 3-32) that iterates through all branches of the index tree. For the first iteration, the first index bucket is downloaded (line 5-6). If the bucket is the root node of the index, then there is no sibling branch and $next$ remains $null$; otherwise, $next$ is set to the time of the next index segment(line 10-12), which contains the next branch.

In the inner loop (line 13-31), the algorithm pops the next data item, a bucket, to be downloaded, and downloads the data item (line 15). If the bucket downloaded is index, then each index entry, expressed as (mbr, $time$) pair, is push onto the stack (line 17-22). If bucket is not index, but is data, then the following are performed (line 23-30).

1. Computes local skyline, S', of the data bucket (line 24).

2. Computes local pruning regions, R', from S' (line 24).

3. Prune records in S and stack entries covered by R' (line 26-27).

4. Add (union) local pruning regions, R', to global regions, R (line 28).

5. Add (union) local skyline,S', to global skyline, S (line 29).

Note that S are candidate skyline since there could be data objects broadcast later that dominate the earlier candidate Skyline points. For example, if a later candidate point s' dominates an earlier point s, then s is removed from S and the pruning region is enlarged by the later point s'.

The skyline computation phase ends when the stack is empty and S is returned as skyline (line 33).

Algorithm 1 Record-Based Skyline(σ)

```
1.  S ← ∅
2.  R ← ∅
3.  repeat
4.     next ← null
5.     if first iteration then
6.        bucket ← search first index bucket
7.     else
8.        bucket ← GetBucket(next)
9.     end if
10.    if not IsRootNode(bucket) then
11.       next ← bucket.time_of_next_index_segment
12.    end if
13.    while not stack.empty or first iteration do
14.       if not first iteration then
15.          bucket ← GetBucket(stack.pop)
16.       end if
17.       if IsIndex(bucket) then
18.          for E_i in bucket, i = b to 1 do
19.             if not Covers(R, E_i) then
20.                stack.push(E_i)
21.             end if
22.          end for
23.       else
24.          S' ← Skyline(bucket, σ)
25.          R' ← GetPRegions(S', σ)
26.          Prune(S, R')
27.          Prune(stack, R')
28.          R ← R ∪ R'
29.          S ← S ∪ S'
30.       end if
31.    end while
32. until next = null
33. return S
```

2.5.2 Index-Based Skyline Pruning

The drawback of the RPS is that the algorithm has to receive data records before pruning regions can be formed. In this section we present index-based pruning skyline (IPS) that utilizes the index bounding boxes to form pruning regions. The expectation is that the index usually precede data records. If we can make pruning regions early using index, then more of the index tree could be pruned. Note that this algorithm only produces correct result when the bounding boxes, of the same level of the index tree, do not overlap, such as index structures described in [81] [21] [4]. We define the follow definition and theorem to formulate IPS:

Definition 2.5.3 (Anti-Dominant Point)**.** *Given a n-dimensional rectangle, B, and a list of skyline preference specifiers, σ, the anti-dominance point of the rectangle $ad(B, \sigma)$ is the point of intersection of all "worst" bounds of B, where the "worst" is defined by σ.*

Theorem 2.1. *Given that bounding boxes on the same level of the index do not overlap and the following:*

- *A n-dimensional minimal bounding box, B*
- *Skyline preference, σ*
- *Dominant point of B, $d = d(B, \sigma) = (d_1, d_2, .., d_n)$*
- *Anti-dominant point of B, $a = ad(B, \sigma) = (a_1, a_2, .., a_n)$*

a set of n pruning regions, R, can be created from B, as

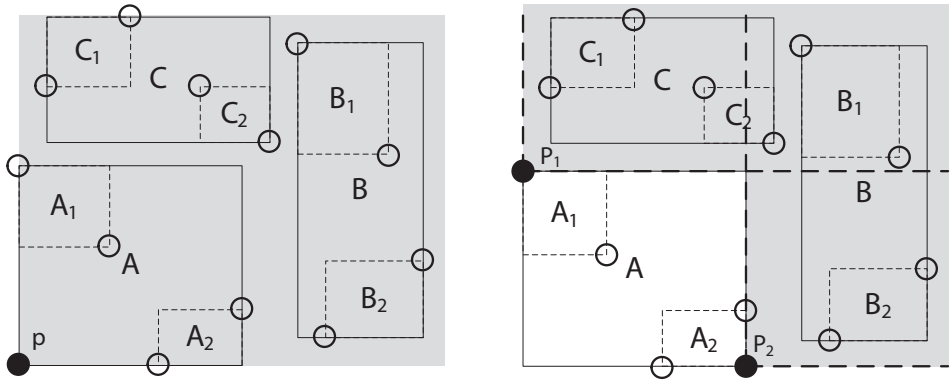
$$R_B = \{R((a_1, d_2, .., d_n), \sigma), R((d_1, a_2, .., d_n), \sigma), .., R((d_1, d_2, .., a_n), \sigma)\} \tag{2.7}$$

Proof. According to the definition of minimal bounding box [75], for each bound of B, there must be a point that defines that particular bound, such that there is a set of points, $P = \{p_1, p_2, .., p_m\}$ $m \leq 2 \times n$, that defines B. By Property 2.5.1, a set of pruning regions can be created for each

point in P, R_P = $\{R(p_1, \sigma), R(p_2, \sigma), .., R(p_m, \sigma)\}$. Since R_B is created from the anti-dominant point of B, R_P completely covers R_B. □

Theorem 2.1 states that for each minimal bounding box, B, of the index that we have examined, a set of n pruning regions, R_B, is created. We can improve memory usage slightly by merging all pruning regions in R_B and create only one, R_m = $R(d(B, \sigma), \sigma))$, to keep in memory. Not only does R_m cover R_B, but it also covers the entire B; therefore, if this optimization is used, the algorithm must keep track that B should still be considered for skyline computation if it had not already visited.

We term the original n-region and the optimization one-region strategies. Figure 2.9(a) illustrates the one-region strategy in which one pruning region is created from minimal bounding box A. Note that A is inside the pruning region, but should still be visited. Figure 2.9(b) illustrates n-pruning regions created from A. n always equals the dimensionality of the data, in this case n = 2. Our algorithm implements the n-region strategy described by the theorem.



(a) Pruning with one Pruning Region (b) Prune with n Pruning Regions

Figure 2.9: Index-Based Pruning Strategies (min, min)

IPS is shown in Algorithm 2. The initialization and outer loop is the same as RPS. During skyline computation phase (line 13-30), the next expected item (index or data bucket) is popped from the stack and downloaded from the channel. If the item is an index, then each inner (bounding box, broadcast time) pair is checked to see if R covers the bounding box (line 9). If the bounding

box is not covered, then the pair is pushed onto the stack, and n pruning regions are computed as follows:

1. The next level bounding box is pushed onto the stack to be downloaded in future iteration (line 20).

2. The local pruning region, R' is computed from the next level bounding box (line 21).

3. Candidate skyline and queue entries that are covered by R' are removed (line 22-23).

4. Local pruning regions, R', is added to the global pruning regions (line 24).

If the downloaded item is a data bucket, then skyline is computed from the bucket and added to the set of candidate skyline (line 28). As in RPS, when the stack is empty, S is returned as skyline result.

2.5.3 Analysis

In this section we consider tuning time and access latency of the two pruning region skyline algorithms we presented in this section. We assume that the client makes skyline queries at the beginning of a cycle. Since the amount of time spent listening to the channel is the amount of time required for a client to get all the desired data, the tuning time is equal to access latency; therefore, we use the same analysis for both quantities.

We first consider the tuning time for the algorithm that uses the record-based pruning strategy. The best case is that the client gets all desired data from the first data segment, and the rest of the cycle is pruned. In this case, $\beta = h \times \eta + \varsigma$. The expected case is when the client has to listen to half of the program and $E(\beta) = \frac{1}{2}(\iota + \theta)$

We now consider the same quantity for the index-based pruning skyline algorithm. The assumption is the same as before, but a client does not have to traverse the index to the leaf level before they start pruning. On average, a client would have to traverse half of the index tree but would still have to download half of the data; therefore $E(\beta) = \frac{1}{4}\iota + \frac{1}{2}\theta$

Algorithm 2 Index-Based Skyline(σ)

```
1.  S ← ∅
2.  R ← ∅
3.  repeat
4.    next ← null
5.    if first iteration then
6.      bucket ← search first index bucket
7.    else
8.      bucket ← GetBucket(next)
9.    end if
10.   if not IsRootNode(bucket) then
11.     next ← bucket.time_of_next_index_segment
12.   end if
13.   while not stack.IsEmpty or first iteration do
14.     if not first iteration then
15.       bucket ← GetBucket(stack.pop)
16.     end if
17.     if IsIndex(bucket) then
18.       for E_i in index, i = b to 1 do
19.         if not Covers(R, E_i) then
20.           stack.push(E_i)
21.           R' ← GetPRegions(e.mbr, σ)
22.           Prune(S, R')
23.           Prune(stack, R')
24.           R ← R ∪ R'
25.         end if
26.       end for
27.     else
28.       S ← S ∪ Skyline(bucket, σ)
29.     end if
30.   end while
31. until next = 0
32. return S
```

2.6 Experimental Evaluation

In this section, we report our simulation results and evaluate the efficiency of TDI (our program allocation algorithm) and point-based and index-based skyline computation algorithms discussed in previous sections of this chapter. Our simulations are implemented in C# with .NET Framework 4 (compatible with version2) and conducted on a machine of 3.4 GHz Intel Pentium 4 processor, 3 GB RAM, and Windows 7.

The simulations are ran with synthetic data-sets categorized as follows:

- Uniformed: data uniformly distributed in the data space of each attribute.
- Rising: the attributes of the records are correlated to the first attribute of the data-set.
- Falling: the attributes of the records are inversely correlated to the first attribute of the data-set.

In addition, we also conducted simulations of data-sets with varying *record count* and *dimensions*. Record count ranges from 20,000 to 100,000, and dimensions ranges from 2 to 10. Our simulation are memory-only: at the started of a simulation, a data-set is loaded into memory from disk. Table 2.4 lists the size of data elements used in the implementation of the tuning time and index percentage simulation.

Table 2.4: Simulation parameters.

Item	**Size in Bytes**
Pointer of index (ptr)	4
Field of record (f)	8
Record/point (p)	$f \times n$
Minimal bounding rectangle (MBR)	$2 \times p$
Index entry (E)	MBR + ptr

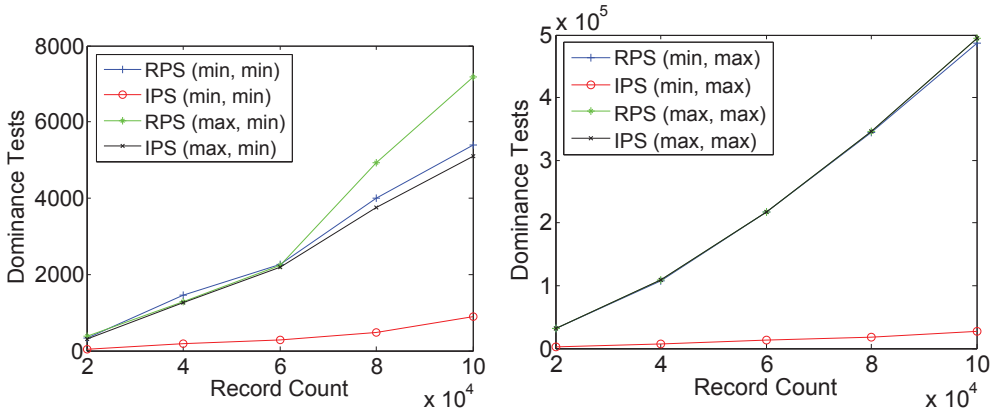
2.6.1 Dominance Tests

Dominance tests measures the number of record comparisons the client has to perform to evaluate a skyline query via Block-Nested-Loop (BNL) [6]. A comparison determines if a record dominates another record. Comparisons are performed when a data bucket is downloaded. Intuitively, as the number of records in the a data-set increases, the number of dominance tests also increases.

Figure 2.10 compares the the number of dominance tests of RPS and IPS with increasing data-set size, from 2×10^4 to 10×10^4 data records (with n = 2 and b = 10). 2.10(a) shows that both algorithms performs well. For the simulation with 6×10^4 records, only 4000 comparisons

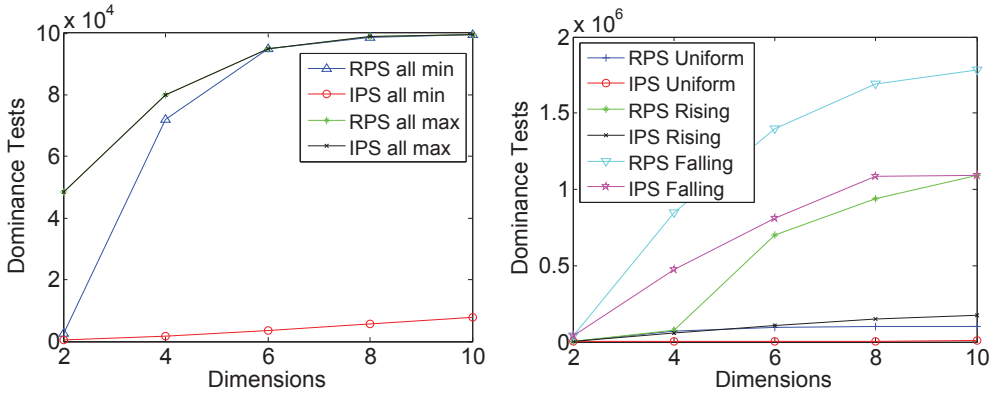
are performed. 2.10(a) shows that when skyline preference is "opposite" of the ordering of data, the pruning is not as effective. For record count of 6×10^4, 2×10^5 comparisons are performed.

Figure 2.11 compares the number of dominance tests with increasing data dimensions. The experiments are conducted with the record count = 10,000, and a b = 10. Figure 2.11(a) compares the results when skyline query attribute specifiers are all min and all max. Figure 2.11(b) shows the result of the number of dominance tests with different data types. The two figures show that as the number of data dimensions increases, the volume (or space) of the data also increases. This leads to more space for the records to "hide" and not fall into the pruning region, and therefore the number of dominance tests increases.



(a) (min, min) and (max, min) (b) (min, max) and (max, max)

Figure 2.10: Dominance Tests vs. Record Count. d = 2, b = 10, Data = uniformed



(a) All Min and All Max. Data = uniformed (b) Mixed Data Types

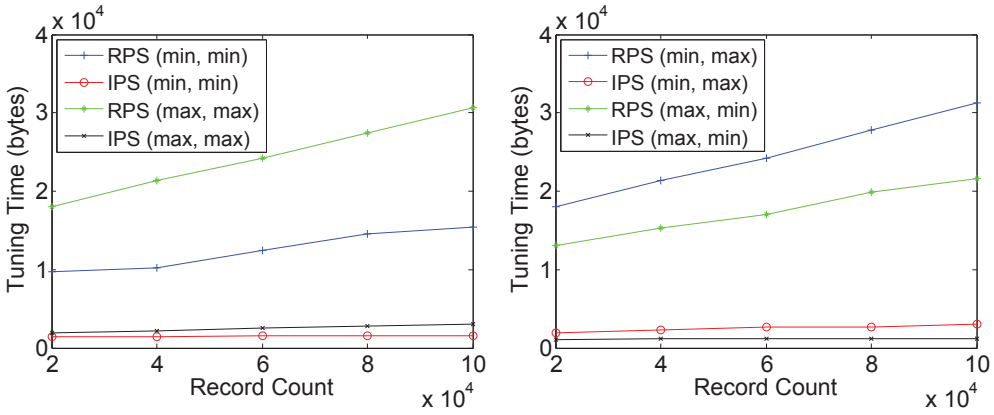
Figure 2.11: Dominance Tests vs. Dimensions. rc = 10000, b = 10

2.6.2 Tuning Time

As discussed in Section 2.2.2, tuning time is the total amount of data the client has to download to fulfill a skyline query. It is measured in bytes. The experiment simulates the creation of the TDI broadcast program by the server. Tuning time is found by simulating a client evaluation of a skyline query from the beginning of the cycle.

Figure 2.12 illustrates tuning time versus increasing record count for all combinations of min and max attribute for 2-dimensional data. In all cases, the IPS performs several factors better than RPS.

Figure 2.13 illustrates the simulation result for tuning time with increasing data dimension. The experiments are run with a record count = 10,000, and a b = 10. Although the number of records stays the same, each additional dimension or data attribute of the data set adds space complexity to the data set. With increasing dimensionality, the cycle length also increases, as well as tuning time.

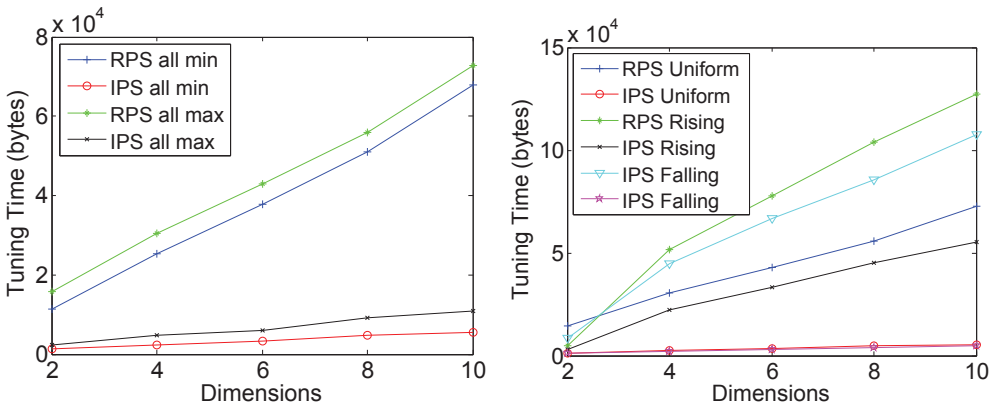


(a) (min, min) and (max, max) (b) (min, max) and (max, min)

Figure 2.12: Tuning Time vs. Record Count. d = 2, b = 10

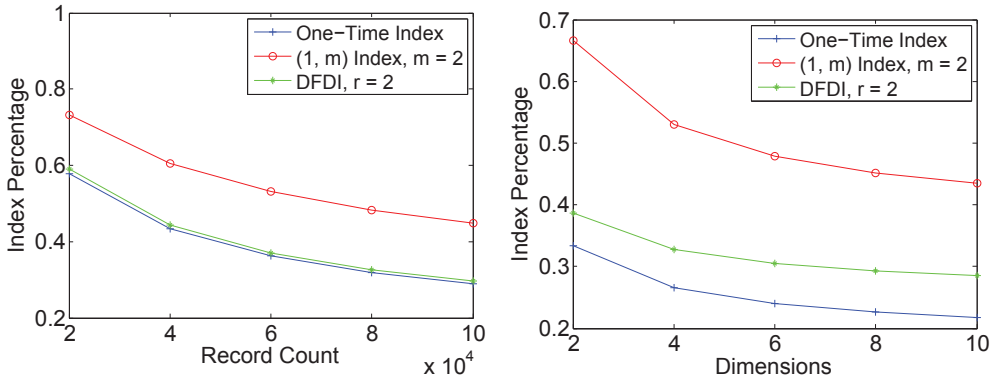
2.6.3 Index Percentage

Index percentage measures the efficiency of the TDI broadcast program allocation technique under increasing record count and increasing data dimension. Index percentage is defined in section 2.2.2.



(a) All Min and All Max (b) Mixed Data Types

Figure 2.13: Tuning Time vs. Dimensionality. rc = 10000, b = 10



(a) IP vs. Record Count (b) IP vs. Dimensionality

Figure 2.14: Index Percentage. b = 10

Figure 2.14(a) shows index percentage versus increasing record count. For the TDI, the simulation is run with a replication level of 2, which means the root and the first level below the root are replicated. For the (1, m) index, m = 2, meaning the complete index is duplicated 2 times in the broadcast cycle. The figure shows that the overhead of the index is high when the number of records is low, but the overhead flattens as the number of records grows. The one-time index is the baseline, and as expected has the lowest index percentage. Although TDI replicated the index for 2 levels, its index percentage is only slightly (16%) higher than the one-time index. This shows that TDI is efficient in terms of space overhead, whereas the (1, m) index has a far worse overhead for only 2 duplications.

Figure 2.14(b) shows index percentage over increasing data dimension. The experiment is conducted with 10,000 records and a branching factor of 10. As the data size grows with the number of dimensions, the index is only slightly affected by the growth. The size of the index grows because it needs extra space to store the information of the extra dimensions, but the tree height is largely unaffected; thus gives the falling of index percentage with growing dimension.

Chapter 3

Hierarchical Data in Relational Database Management Systems

3.1 Introduction

Hierarchical structures (or trees) frequently occur in information management that required to be stored and retrieved. Common examples of such structure include organization chart, in which positions of an organization are displayed as a tree, and hierarchical file system where files are stored in levels of directories.

Due to the popularity of relational database management systems (RDBMS), it is often required to store hierarchy data in such databases. The first apparent question is how to store hierarchical data in tables of RDMBS, which are great for storing a list of items without much design foresight for storing hierarchical data?

This chapter explores ways of storing hierarchical data in relational databases. The goal is to define a few common operations of hierarchical data, the issues of implementing tree structure in relational model, and optimization for these operations.

3.2 Definition

3.2.1 Tree

Hierarchical data structure is also known as a tree, which is defined as a connected acyclic digraph $G = (V, E)$ of a set of vertices V (also called nodes) and a set of edges E that denotes the relationship between two nodes[1]. Figure 3.1 depicts a typical tree.

Each node of the tree can hold any number of data item. In the figure above, only identifiers (A–J) of the nodes are shown for simplicity of illustration. The edges of the tree between two

[1] https://en.wikipedia.org/wiki/Tree_(data_structure)

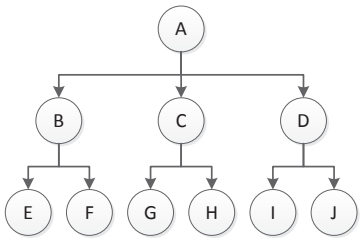


Figure 3.1: Example of a tree.

nodes are often named children (and sometimes parent) relationship. The relationship is often non-symmetric (as in digraph); for example, B is a child of A, but the reverse is not true.

In many literatures of computer data structures, the edges are defined as 'children of' a node. For example, node A has children B, C, and D. In the sense of relational database, it does not matter if the edges are defined as 'children of' or 'parent of'; when one is defined, the other is usually inherited, especially when indexes are used on the relational table in which the tree is stored. The relationship, therefore, will be left for the design of database schema.

3.2.2 Relational Database

The relational data model is proposed by Edgar Codd in 1970 [13]. In this model, a database consists of a set of tables (also known as relations), which each consists of a set of tuples. In colloquial usage, tuples are also called records.

Figure 3.1 illustrates a relational database with 2 tables, each 3 attributes. The teachers table has 4 records (tuples) and the Courses table has 6.

Table 3.1: A Relational Database

(a) Teachers Table

ID	Name	Office
1	James Madison	A
2	John Adams	B
3	Ronald Reagan	C
4	Franklin Pierce	D

(b) Courses Table

ID	Name	Teacher
1	Data Structures	1
2	Intro to Algorithms	1
3	Software Construction	2
4	Computer Networks	3
5	Operating Systems	4
6	Compilers	2

3.2.3 Common Tree Operations

In this section, we define common operations on trees. We only define data reading operations due to the fact that these operations are the most commonly performed operations in database system after the initial insertion of data. Modification and deletion are less common and are usually trivial cases (compared to a myriad of important reading operations) to correct small mistakes. In addition, before updating or deletion, reading must be performed to locate the correct data to modify. With respect to hierarchical data, once an hierarchy is inserted into a database, the structure is rarely changed without removing an entire subtree.

We define the operations in this section without considering how the trees are implemented in relational model. Section 3.4 and 3.3 will cover implementation details and how these operations can be evaluated under a relational designs.

Operations thtat will be used in this chapter are enumerated below. All examples use the tree of Figure 3.1.

1. $Root(n) \rightarrow r$: given a node n, this operation finds the root r of the tree in which n resides: $Member(n, r) \rightarrow true$.

$$Root(A) \rightarrow A$$

$$Root(E) \rightarrow A$$

2. $Parent(n) \rightarrow p$: Given a node n, this operation finds the parent node p, or $null$ if n is a root node.

$$Parent(A) \rightarrow null$$

$$Parent(B) \rightarrow A$$

3. $Children(n) \rightarrow \{c_1, c_2, ..., c_n\}$: Given a node n, this operation finds it's child nodes $\{c_1, c_2, ..., c_n\}$. If n is a leaf node (no children), then the empty set is returned.

$$Children(A) \rightarrow \{B, C, D\}$$
$$Children(E) \rightarrow \{\}$$

4. $Siblings(n) \rightarrow \{n_1, n_2, ..., n_n\}$: This operation finds a list of nodes $\{n_1, n_2, ..., n_n\}$ that have the same depth as node n. If this node has no siblings, the empty set is returned.

$$Siblings(A) \rightarrow \{\}$$
$$Siblings(B) \rightarrow \{C, D\}$$

5. $Leaves(n) \rightarrow \{n_1, n_2, ..., n_n\}$: This operation returns a set of leaf nodes $\{n_1, n_2, ..., n_n\}$ of the subtree rooted at n

$$Leaves(A) \rightarrow \{E, F, G, H, I, J\}$$
$$Leaves(B) \rightarrow \{E, F\}$$
$$Leaves(E) \rightarrow \{E\}$$

6. $Height(n) \rightarrow \mathbb{N}$: Given a node n, this operation returns the count of node of longest path from n to a leaf node.

$$Height(A) \rightarrow 3$$

$$Height(E) \rightarrow 1$$

7. $Depth(n) \rightarrow \mathbb{N}$: Given a node n, this returns $|Path(n)|$.

$$Depth(A) \rightarrow 1$$

$$Depth(E) \rightarrow 3$$

8. $Path(n) \rightarrow (n_1, n_2, ..., n_n)$: given a node n, this operation retrieves the sequence of nodes $(n_1, n_2, ..., n_n)$ that forms the path from root to node n, where n_1 is the root and n_n is n. This operation is the same as retrieving all *ancestors* of n.

$$Path(A) \rightarrow (A)$$

$$Path(E) \rightarrow (A, B, E)$$

9. $Member(n, t) \rightarrow boolean$: given two nodes n and t, this operation tests if node n is a *descendant* of node t – or, if n is in the subtree rooted at t.

$$Member(B, A) \rightarrow true$$

$$Member(A, B) \rightarrow false$$

$$Member(C, B) \rightarrow false$$

10. $Tree(n) \rightarrow (n_1, n_2, ..., n_n)$: given a node n, this operation returns a sequence of nodes $(n_1, n_2, ..., n_n)$ obtained by breadth-first traversal (BFT) from n, where $n_1 = n$. The original tree should be possible to be reconstructed from this sequence.

$$Tree(A) \rightarrow (A, B, C, D, E, F, G, H, I, J)$$

$$Tree(B) \rightarrow (B, E, F)$$

3.3 Adjacency List

The simplest way of representing hierarchy in relational data model is a method called adjacency list. In this approach, a table (we will call it Nodes table) is used to store the nodes of one or multiple trees, and each record of the table represent a node.

Each record of the Nodes table consists of field of primary key of the identifier of the node, and a field of foreign key that holds the identifier of the parent node, which references the same table. The root node of a tree would have *Parent* = *null*. The schema for the adjacency list table is shown in Listing 3.1.

Note: The name 'adjacency list' may cause a bit of confusion. An adjacency list implies a list of child (adjacent) nodes of a node, but the schema for the table above only has a reference from a node its parent node, which should probably more aptly named 'parent-link' approach.

Listing 3.1: Schema for adjacency list

```
1 CREATE TABLE Nodes (
2   ID CHAR PRIMAY KEY NULL,
3   Parent CHAR NOT NULL,
4   FOREIGN KEY (Parent) REFERENCES Nodes(ID)
5 )
```

Listing 3.2: SQL for querying child nodes.

```
1 SELECT * FROM Nodes WHERE Parent='A'
```

As noted in section 3.2.1, this distinction is actually not important in relation database. The reason is that with a reference to parent defined, the *children* of a node (adjacency list) can be easily found. For example the child nodes of A can be by found using the query in Listing 3.2:

3.3.1 Operations

In this section, we present how the operations defined in Section 3.2.3 can be implemented in the context the adjacency list model. For operations that involve simple queries, we provide only the queries; for operations that requires more complex operations (e.g. loops), we present the algorithm in Java programming language.

Root Operation

If the Nodes table contains only a single hierarchy, the operation of finding the root of an arbitrary node in the tree would be a trivial query of looking for the record whose Parent attribute is null. This is *not* the case for us. We assume that the Nodes table contains records for multiple trees. This complicate the query quite a bit and requires us to use some form of iteration (or recursion) to traverse to the top of the tree (until we reach a node with Parent = null). This procedure makes the time-complexity of operation $O(d_n)$. Listing 3.3 shows the code for this operation.

Listing 3.3: Root for Adjacency List

```
1  /** Returns the first node after executing the query.*/
2  static Node getOneNode(String query){...}
3
4  /** Returns the root node of node n. */
5  static Node Root(Node n) {
6    if(n.Parent == null)
7      return n;
8    Node parent = getOneNode("SELECT * FROM Nodes WHERE ID='" +
       n.ID + "'");
9    return Root(parent);
10 }
```

Parent, Children, Siblings Operations

In the adjacency list model, these operations are trivial and can be implemented via a short query (Listing 3.4). The Parent operation (line 6) is trivial because every node contains a foreign key that points to the primary key of the parent node. Since the primary key indicates the exact location of a record in the database, retriving a parent node is $O(1)$ operation.

For the children (line 9) and siblings (line 12) operations, if the 'parent' field is indexed using B-Tree [14], then the time-complexity of this query is $O(\log |N|)$, where $|N|$ is the total number of rows in the Nodes table. If the field is indexed using a hash index, the time-complexity for these becomes $O(1)$.

Leaves, Height, Depth, Path, Member Operations

The algorithms for finding leaves and height of a tree will be similar. Given a node, both algorithms iteratively or recursively perform the same operation on the children of the node until the iteration reaches a leaf node. For the Leaves operation, when the recursion reaches a leaf node, the node is added to the result and recursion ends. For the Height operation, when a leaf node is reached, the algorithm returns 1. The parent call returns $1 + \max(\text{recursive calls})$.

Listing 3.4: Parent, Children, and Sibling

```
1  --
2  -- Given a node $n$ and n.ID='E'
3  --
4
5  -- Get the parent of node E
6  SELECT * FROM Nodes WHERE ID=(SELECT Parent FROM Nodes WHERE
       ID='E')
7
8  -- Get the children of node E
9  SELECT * FROM Nodes WHERE Parent='E'
10
11 -- Get the siblings of node E
12 SELECT * FROM Nodes WHERE Parent=(SELECT Parent FROM Nodes
       WHERE ID='E') AND ID != 'E'
```

Note that for every node in the tree, a call to Leaves() is made. This made the time complexity of these operations $O(|N|)$. This is very inefficient. Considering a tree with 10,000 nodes, which will result in 10,000 queries being made to the database.

We only show the algorithm for Leaves in Listing 3.5.

In addition, the algorithm for Depth, Path, and Member are very similar to the algorithm for finding root. Each algorithm recursively (or iteratively) reaches for the root of the tree while keeping state. For example, the algorithm for depth has to keep track of node count during the iterations. The complexity is the same as finding the root, which is $O(d_n)$ which is proportional to $O(\log |N|)$.

For the algorithm for Memeber, the iteration can stop once the root node of the subtree t is reached. The time-complexity of the implementation is $O(d_n)$. The algorithm is given in Listing 3.6.

Tree Operation

Listing 3.7 shows the algorithm to get the BST of tree in this relational model.

Listing 3.5: Leaves

```
1  /** Returns an array of leaf nodes under n. */
2  static Node[] Leaves(Node n) {
3    ArrayList<Node> result = new ArrayList<Node>();
4    Leaves(n, result);
5    return result.toArray();
6  }
7
8  /** Recursive method.*/
9  static void Leaves(Node n, List<Node> result) {
10   Node[] children = getChildren(n);
11   if(children.length == 0) {
12     result.add(n);
13     return;
14   }
15   for(Node child : children)
16     Leaves(child, result);
17 }
```

Listing 3.6: Member

```
1  static boolean Member(Node n, Node t) {
2    while(n != null) {
3      if(n.equals(t))  //n = t
4        return true;
5      n = getParent(n);
6    }
7    return false;
8  }
```

Listing 3.7: Tree via Adjacency List

```
1  static Node[] Tree(Node n) {
2    Queue<Node> q = new LinkedList<Node>();
3    ArrayList<Node> result = new ArrayList<Node>();
4    q.add(n); //enqueue
5    result.add(n);
6    while(!q.isEmpty()) {
7      Node head = q.remove();
8      Node[] ch = getChildren(head);
9      for(Node c : ch) {
10       result.add(c);
11       q.add(c);
12     }
13   }
14   return result.toArray();
15 }
```

After examining these implementations, we come to the conclusion that the operations are trivial in the adjacency list model are: *1)* parent, *2)* children, *3)* siblings.

All other operations require iterations (or recursions): *1)* root, *2)* leaves, *3)* height, *4)* depth, *5)* path, *6)* member, *7)* tree. The number of queries of these operations usually depend on the depth of the node in question. Each query submitted to the database incurs communication cost between the processing program and the database; therefore, the cost of computing these operations increases as the depth of the tree increase. The time-complexity of this implementation is $O(|N|)$.

3.4 Nested Sets

Another way of representing hierarchical data in relational database is the nested set model [50]. This model is an extension of the adjacency list model. The goal of this model is to eliminate the need to issue multiple queries to the database (and the use of recursion) by placing ordering on the nodes. Listing 3.8 shows the relational schema for the nested set model.

Listing 3.8: Nodes table for nested set model

```
1  CREATE TABLE Nodes (
2    ID CHAR PRIMARY KEY NOT NULL,
3    Parent CHAR NULL,
4    L INT NOT NULL,
5    R INT NOT NULL,
6    FOREIGN KEY (Parent) REFERENCES Nodes (ID)
7  )
```

Listing 3.9: Algorithm for Left-Right value assignment for nested set model

```
1  int AssignLR(Node node) {
2      return AssignLR(node, 1);
3  }
4
5  /** Recursively assign LR for each node */
6  int AssignLR(Node node, int count) {
7    node.L = count;
8    Node[] children = getChildren(node);
9    for(Node child : children)
10     count = AssignLR(child, count + 1)
11   node.R = ++count;
12   return count;
13 }
```

In this model, there are, for each node, two additional fields: L and R. These two fields stand for left and right. These two values provides an ordering of nodes in a tree. Algorithm for assigning L and R value for nodes is shown in Listing 3.9.

The heart of the algorithm in Listing 3.9 is the recursive version (line 6–15) of $AssignLR()$ function. This function takes 2 parameters: $node$, which is the root node of a tree, and $count$ which is a variable to keep track of the current LR value. The $count$ is initially set to 1, so that L of the root is set to 1.

The key points the algorithm is given below:

1. Initially, the function sets L of the current node as the initial value of $count$ that is passed in (line 9). For example, if $node$ is the root node, and initial $count = 1$, then the left value of the root will be 1.

2. For each child of $node$, perform depth-first traversal (DFT) by recursively call $Asssign()$ function (line 11–12). The calls traverse the tree (down-trip) from the root the leaf nodes, and at every call $count$ is incremented.

3. On the up-trip (line 13), R of $node$ is assigned $count + 1$. If node is leaf, then R is assigned, $L + 1$. Else, R is assigned R of last child + 1.

Figure 3.2 shows the left and right values for the tree in the Figure 3.1 after executing $AssignLR()$.

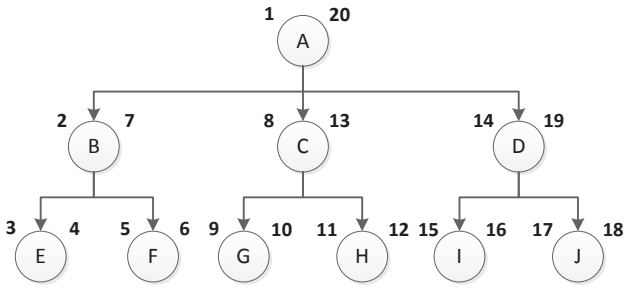


Figure 3.2: Example of a tree in nested set model.

The left and right (LR) values puts an order on the nodes in the database and establish the following property.

Property 3.4.1. *Given two nodes a and d, d is a descendant of a if $(a.L < d.L)$ and $(a.R > d.R)$.*

3.4.1 Operations

In this section we present how the operations defined can be implemented in the nested sets model. Due to the fact that the nested sets model is an extension of the adjacency list model, implementation of all operations still work in this model; however, the performance (in terms of

Listing 3.10: Nested Sets Leaves Operation

```
1 SELECT * from Nodes N WHERE L > 2 AND R < 7 AND NOT EXIST (
    SELECT * FROM Nodes WHERE Parent=N.ID)
```

time of query) of some operations can be improved taking advantage of the left and right values provided by this model.

The simple operations in the previous model cannot be improved further. These operations are Root, Parent, Children and Siblings. We use the same code as provided earlier.

Other operations we can improve efficiency using this model. In previous section, we use iterations (loops or recursions) to realize some of the operations, where in each iteration, one query is submitted to the database. Using nested sets model, in most cases however, we can reduce iterations used in the adjacency model to a single query. The operations we can improve include Leaves, Height, Depth, Path, Member, and Tree.

Leaves Operation

The Leaves operation no longer require traversal to the bottom of the tree and more importantly required to visit every node in the tree. With the structural information provided by the LR values, we can find the leaf nodes by looking for nodes with the correct LR boundaries that do not have child node. Listing 3.10 shows the query to accomplish, given an sample input node of $\{ID : B, L : 2, R : 7\}$.

The illustrated query is a correlated query[2]. The outer query (select) simply limits the selected nodes to be the descendants of input node. For each selected node in the outer query, the subquery checks if there is any child node.

The outer query can be accomplished through two index scans, with the time-complexity of $O(2 \log |N|)$. The inner query is the same as the children operation discussed in section 3.3.1, which in worst case is also $O(\log |N|)$. With the total complexity of $O(2 \log^2 |N|)$, which has a growth rate similar to $O(\log |N|)$.

[2]http://en.wikipedia.org/wiki/Correlated_subquery

Listing 3.11: Path, Depth, Member Operations

```
1 -- Path Query
2 SELECT * FROM Nodes WHERE L <= 3 AND R >= 4 ORDER BY L
3
4 -- Depth Query
5 SELECT COUNT(*) FROM Nodes WHERE L <= 3 AND R >= 4
```

Path, Depth, Member Operations

In addition to the implementation illustrated in Section 3.3.1, we can also evaluate these operations using the nested sets model, which can be directly obtained from Property 3.4.1. Path is simply a list of ancestors, which we can easily obtain, and the query node itself. Similarly, Depth is the count of the number of records returned from Path Given an input node of $\{ID : E, L : 3, R : 4\}$, Listing 3.11 shows the query for Path(line 1-2), and Depth(line 4-5).

The implementation presented in Section 3.3.1 (using pure adjacency list) and the implementation presented in this section (using LR-values) both have the time complexity of $O(\log |N|)$. For the pure adjacency list implementation, this is easy to see because the number of nodes to search is simply the depth, which is proportional to $log\ |N|$. As for the implementation in this section, the query evaluator will have to traverse the index tree for the L and R fields to find the qualifying nodes, which also produce $\log$ time-complexity.

Though both produce similar analytical time-complexity, there are differences between the two approaches. The first difference is that the query presented in this section performances the entire evaluation on the database server and sends the final result back to the client; whereas, the logic of the approach of Section 3.3.1 is located on the client, therefore, the server has to send intermediate results back to the client. In other words, the adjacency list implementation requires multiple queries and and results sent between the client and the server. The second difference is that the nested set approach has to traverse the index tree for both R and L fields. The adjacency list approach does not have this complexity.

Listing 3.12: Member Operation

```
1 static boolean Member(Node n, Node t) {
2     return (t.L < n.L) && (t.R > n.R);
3 }
```

Listing 3.13: Query for Tree for tree A via nested set model

```
1 SELECT * FROM Nested_Set WHERE Left > 1 AND Right < 20 ORDER
    BY left ASC
```

Member Operation

Member match exactly the definition of Property 3.4.1. Listing 3.12 shows a trivial code for this operation.

The time complexity of this operation is $O(1)$.

Tree

Getting the entire hierarchy of a tree is the same as getting all the descendants of the root node (including the root). If the database contains only one tree, then this operation involves retrieving all entries in the nodes table. However, we assume that the database contains multiple disconnected trees.

Listing 3.13 shows this query for the tree in Figure 3.2, in which the left and right value for the root node A is 1 and 20, respectively.

To reconstruct the hierarchical structure from the flat list returned from the query above, we look at the result of the query (of the tree in Figure 3.13) in the Table 3.2 below.

There are two ways to reconstruct the tree, since there are redundant information in the result table to reconstruct the tree. The first way is to only use the 'Parent' column. The second way is to use only the 'Left' and 'Right' columns (not needing the 'Parent' column). Listing 3.14 shows the algorithm for the first approach.

Listing 3.15 shows the algorithm for the second approach.

Listing 3.14: Tree reconstruction using Parent

```
Node ReconstructTree(List<Record> records) {
  Node root, current = null;
  for(Record record : records) {
    if(current == null) {
      current = MakeNode(record);
      root = current;
    }
    else {
      while(current.ID != record["Parent"])
        current = current.Parent;
      Node node = MakeNode(record);
      current.Children.Add(node);
      current = node;
    }
  }
  return root;
}
```

Listing 3.15: Tree reconstruction using Left and Right value

```
//Returns the root node as the tree
Node ReconstructTree(List<Record> records) {
  Node root, current = null;
  for(Record record : records) {
    if(current == null) { //This should only be run once
      current = MakeNode(record);
      root = current;
    }
    else {
      while(!(current.Left < record["Left"] and current.Right
         > record["Right"]))
          current = current.Parent;
      Node node = MakeNode(record);
      current.Children.Add(node);
      current = node;
    }
  }
  return root;
}
```

Table 3.2: Query result of Tree operation

ID	Left	Right	Parent
A	1	20	null
B	2	7	A
E	3	4	B
F	5	6	B
C	8	13	A
G	9	10	C
H	11	12	C
D	14	19	A
I	15	16	D
J	17	18	D

Print Tree

The tree structure can be easily outputted by a program after the hierarchy has been reconstructed with the algorithms in the previous section. However, sometimes a tree has large number of nodes and we may only need to output the tree structure without needing to reconstruct the tree in memory.

This section shows an algorithm to print the ID of the nodes with increasing indention to show the hierarchy of the tree.

3.5 Evaluation

This section presents the performance evaluation of the adjacency list and nested sets models presented in section 3.3 and 3.4, respectively. Performance are evaluated based on the running time of the operations described in section 3.2.3, under each data model.

The evaluation is composed of two parts: the server and the client. The server runs an instance of MySQL database and holds hierarchical structure and data. The client connects to the server via local area network, implements the operations, and reports the amount of time to run them.

This section is organized as follows. Section 3.5.1 and section 3.5.2 describe the environment and configuration of the server and the client.

Listing 3.16: Tree reconstruction using Left and Right value

```
1  void PrintTree(List<Record> records) {
2    //Make new stack to hold right values;
3    //length = 0
4    Stack stack;
5    foreach(Record record in records) {
6      while(stack.top < record["right"] and !stack.isEmpty)
7        stack.pop()
8      //Print indentation
9      for(int i = 0 to stack.length - 1)
10       Print(" ")
11     //Print current record
12     Print(Record.ID)
13     stack.push(record["right"])
14   }
15 }
```

3.5.1 Server Configuration

Table 3.3 shows the software and hardware configuration of the server machine.

Table 3.3: Dataset size

CPU	Intel Processor 64-bit
Memory	4 GB
Operating System	Ubuntu 13.10
Database System	MySQL 5.5 Community Edition

In addition, when running the database system, the following settings are made:

Table 3.4: Database Server Non-default Settings

Variable	Set Value	Default Value
max_sp_recursion_depth	100	0
maximum_stack memory	100	100
some other memory	100	100

Note: briefly describe the reason for above settings.

Listing 3.17: Nodes table for nested set model

```
1 java -cp lib/mysql-connector-java-5.1.29-bin.jar:. -Xms8g -
    Xmx9g Experiment
```

3.5.2 Client Configuration

The client machine is responsible for running the experiment code, which is written in Java and connected to the database using MySQL JDBD Connector[3]. Table 3.5 lists configurations of the client environment.

Table 3.5: Dataset size

CPU	Intel Processor 64-bit
Memory	12 GB
Operating System	Ubuntu 13.10 Desktop
Programming Language	Java 7 (OpenJDK 7)
Database Connector	JDBC mysql-connector-java-5.1.29

At early stages of coding the experiments, the author repeatedly getting Out Of Memory Error from the Java Virtual Machine. The code for the experiments have been rewritten several times to reduce memory consumption. The error can also be mitigated by expanding the allocated memory to the program. The following are the command sets the minimum and maximum heap memory allocated to the experiment to 8 and 9 gigabytes.

3.5.3 Data Format and Generation

The experimental datasets are synthetically generated relational hierarchies, each containing the fields ID, Parent, L and R as described for the Nodes table in Listing 3.8. In addition, each record (an entry in data table) also contains additional 90 ASCII characters, divided into two fields (45 characters each), to simulate data payload in typical database record. However, these additional data is currently not being used in experiments, due to the limitation of amount of memory in machines that run the experiments. The structure of the experimental data shown in Table 3.6.

[3]http://www.oracle.com/technetwork/java/javase/jdbc/index.html

Table 3.6: Experimental data relational format

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
parent	int(11)	YES	MUL	NULL	
l	int(11)	NO	MUL	NULL	
r	int(11)	NO	MUL	NULL	
data1	varchar(45)	NO		NULL	
data2	varchar(45)	NO		NULL	

The datasets are created by our data generation SQL script (see appendix) and stored on in the database on the server. In addition to being a full tree[4], each dataset is characterized by the following properties:

1. Order: the number of children per node. This is also known as the fan-out or out-degree of a tree.

2. Height: the number of nodes of the longest path from root to a leaf node.

Each of the properties ranges from 2 to 10, with the exception that higher-order datasets have lower max height. For each combination of order and height, a dataset is generated and stored in a database table. Table 3.7 shows the size (number of records) of each data table.

Table 3.7: Test data size with respect to Order and Height

		Height								
		2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Order	**2**	3	7	15	31	63	127	255	511	1,023
	3	4	13	40	121	364	1,093	3,280	9,841	29,524
	4	5	21	85	341	1,365	5,461	21,845	87,381	349,525
	5	6	31	156	781	3,906	19,531	97,656	488,281	2,441,406
	6	7	43	259	1,555	9,331	55,987	335,923	2,015,539	12,093,235
	7	8	57	400	2,801	19,608	137,257	960,800	6,725,601	47,079,208
	8	9	73	585	4,681	37,449	299,593	2,396,745	19,173,961	153,391,689
	9	10	91	820	7,381	66,430	597,871	5,380,840	48,427,561	435,848,050
	10	11	111	1,111	11,111	111,111	1,111,111	11,111,111	111,111,111	1,111,111,111

With the server configuration described earlier in this section, an estimate of the amount of time to generate the data sets is shown in Table 3.8.

[4]http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees

Table 3.8: Experimental data generation time estimate

		Height								
		2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Order	**2**	200 ms	470 ms	470 ms	2.07 sec	4.2 sec	8.47 sec	17 sec	34.07 sec	1.14 m
	3	270 ms	870 ms	2.67 sec	8.07 sec	24.27 sec	1.21 min	3.64 min	10.93 min	32.80
	4	330 ms	1.4 sec	5.67 sec	22.73 sec	1.52 min	6.07 min	24.27 min	1.62 hr	6.47
	5	400 ms	2.07 sec	10.4 sec	52.07 sec	4.34 min	21.7 min	1.81 hr	9.04 hr	1.88 d
	6	470 ms	2.86 sec	17.27 sec	1.73 min	10.37 min	1.04 hr	6.22 hr	1.56 days	9.33 d
	7	530 ms	3.8 sec	26.67 sec	3.11 min	21.79 min	2.54 hr	17.79 hr	5.19 days	36.33 d
	8	600 ms	4.87 sec	39 sec	5.20 min	41.61 min	5.55 hr	1.85 days	14.79 days	118.36
	9	670 ms	6.07 sec	54.67 sec	8.20 min	1.23 hr	11.07 hr	4.15 days	37.37 days	336.30
	10	730 ms	7.4 sec	1.23 min	12.35 min	2.06 hr	20.58 hr	8.57 days	85.73 days	857.34

3.5.4 Experimental Result

The operations defined in Section 3.2.3 are divided into three categories based on their running time complexity which are $O(logN)$, $O(N)$, and $O(1)$. Members in a category have the similar running time; therefore we have to show the performance for one operation per group. The membership of each category shows as follows:

1. $O(logN)$: Root, Depth, Path, Member
2. $O(N)$: Heaves, Height, Tree
3. $O(1)$: Parent, Children, Siblings

In this section, we report the performance of category 1 and 2. Category 3 is not reported since the input size does not affect the outcome and that the implementation are the same for both adjacency-list and nested-sets model.

Category 1 - O(log N)

Figure 3.3 depicts the running time in milliseconds of the finding root operation for the adjacency list model. From this result, we can rough infer that the time complexity of this algorithm is linear with respect to the depth, and constant with respect to the width (order) of the tree.

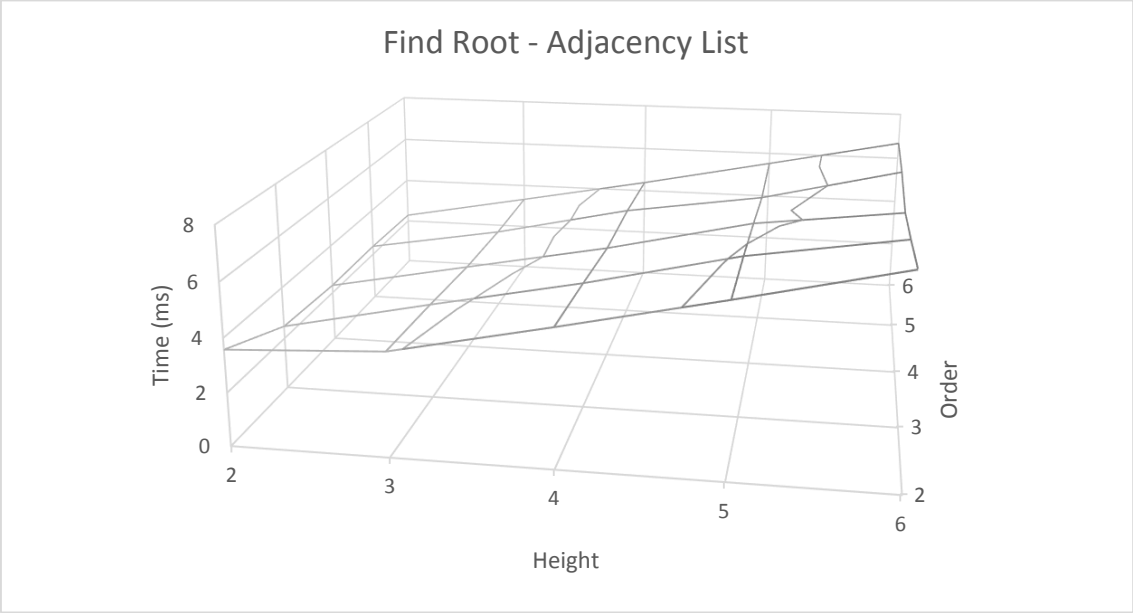


Figure 3.3: Running time using Adjacency List Model.

Figure 3.4 depicts the running time in milliseconds of the finding root operation for the nested sets model. One can easily see that it is an improvement over the adjacency list model. In this result, the time complexity with increasing depth of the tree has reduced to constant. The time complexity with increasing width stayed constant, with slight constant improvement.

If we take a slice of the previous two figures down where height = 8, we can see the growth of the algorithm with respect to order as shown in Figure 3.5. The figures shows, as expected, that the growth of the order of the tree has little effect on the running time of the operation.

For the adjacency list model, the trend appears to be decreasing. However if we look at the running time from order 4 to 8, the trend is flat. I believe that the initial decreasing trend is caused by database warm-up. As more data are cached, the database system is able to reach stable-state.

In addition to the two described models, the figure shows the growth trend for adjacency list model implemented using MySQL stored procedure (labeled Stored). The purpose of the stored implementation is to see if stored procedures facilitates performance. In this case, the store version is better than both adjacency list and nested sets models.

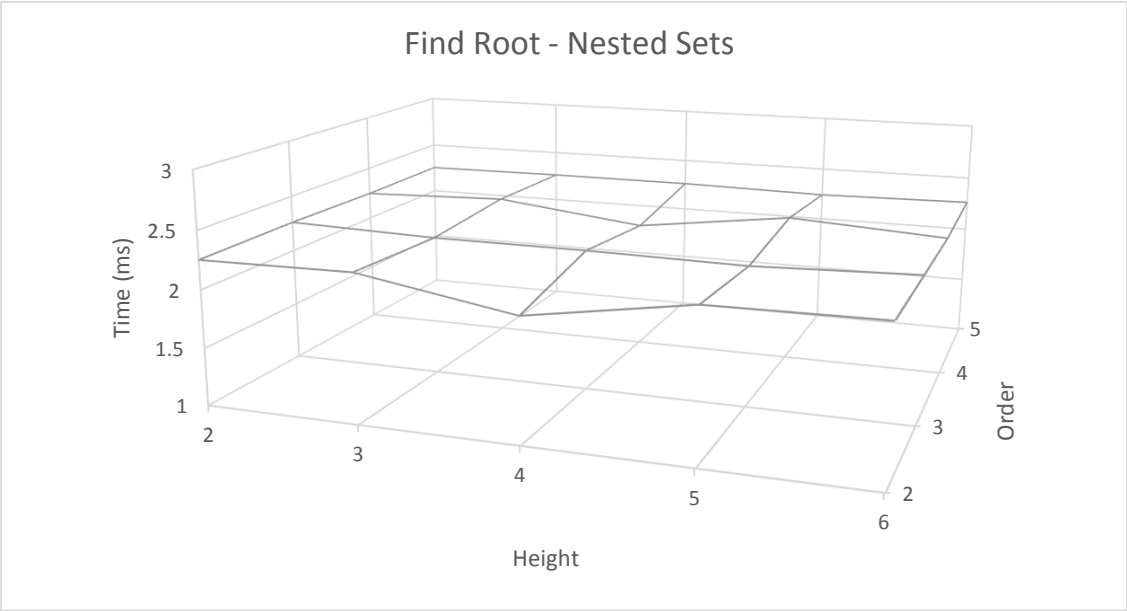


Figure 3.4: Running time using Nested Sets Model.

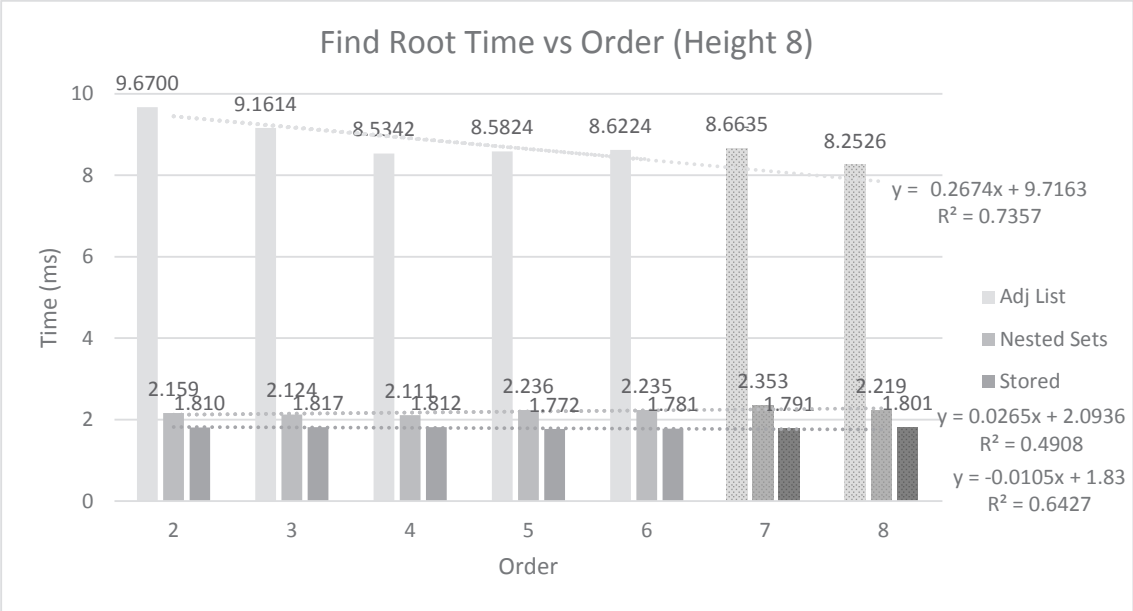


Figure 3.5: Running time vs node out-degree (order) at height = 8.

In the figure, the trend line is draw using linear regression from the data from order 1 to 6. The measurement of order 7 and 8 are draw to verify the correctness of the growth model.

In the same manner, if we take a slice of Figure 3.3 and 3.4 down where order = 8, we can see the growth of the algorithm with respect to height as shown in Figure 3.6.

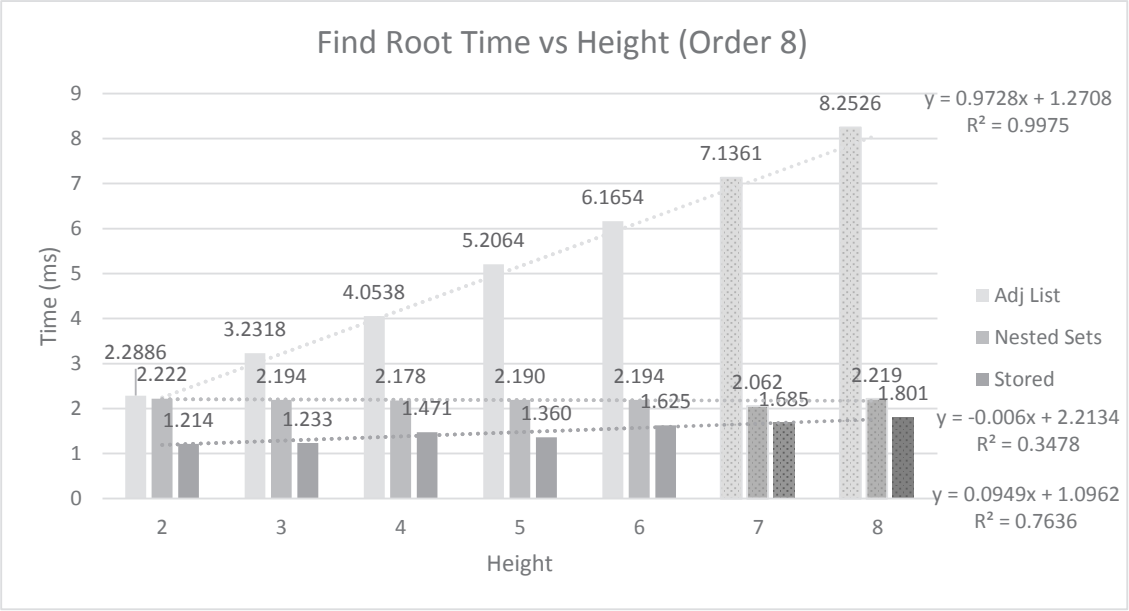


Figure 3.6: Running time vs tree height at order = 8.

The figure shows that for adjacency list model, the running time grows linearly with respect to height. In contract, the running time for the nested sets model shows big improvement over adjacency list, with almost perfect flat trend line model, and indicates constant time-complexity. Once again, the data for the stored model shows that the running time are better than both other models; however, the trend shows a linear growth rate, which is not not better than nested sets model larger data set.

Category 2 - O(N)

Figure 3.7 depicts the running time, in milliseconds, of the finding root operation using the Adjacency List Model. The running time stayed relative flat, below 100 ms for height and order, below 4.

Above 4, the figure shows that the running time appears to be growing exponentially with respect to both dimensions. The exponential growth with respect to height is expected because the

operation has $O(N)$ complexity and each increase of height increase N by a factor of w, where N is the number of nodes and w is the order (or width) of the hierarchy.

However, the exponential growth with respect to the order w in the figure is not expected. Increase in the width of the hierarchy only linearly increase the number of node N. This is likely caused by additional processing (such as index tree traversal) required by the database system that makes the cost of all nodes higher than $O(N)$.

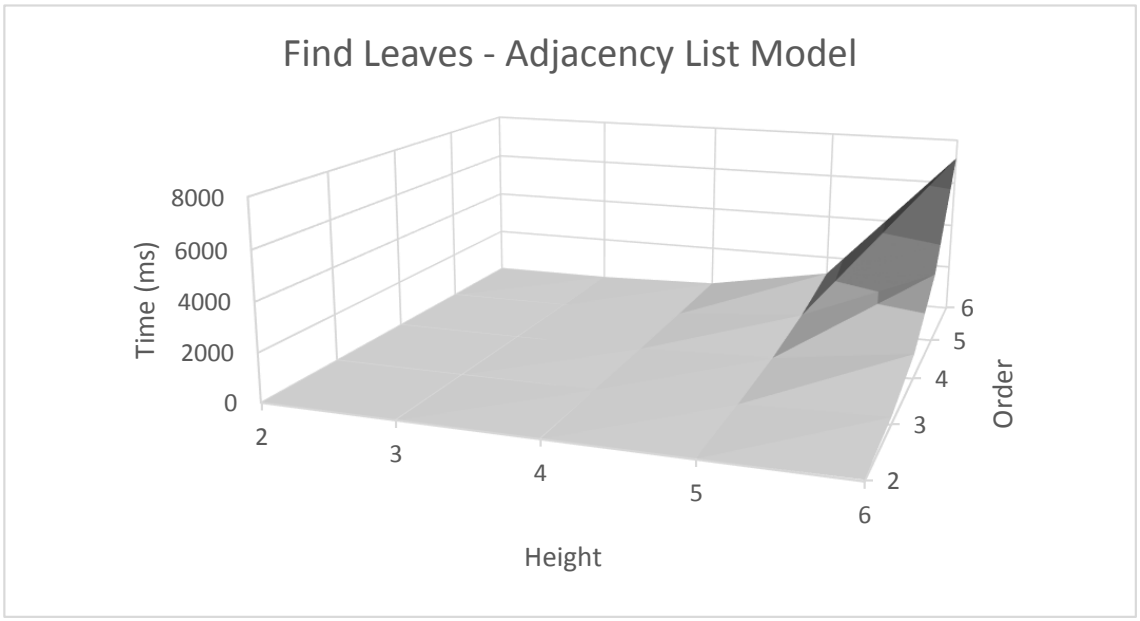


Figure 3.7: Running time of Finding Leaves using Adjacency List Model.

Figure 3.8 depicts the running time, in milliseconds, of the operation using Nested Sets Model. Careful reader will find that the figure looks almost the same as Figure 3.7. Although the two figures have the same shape, the running time for the Nested Sets Model is much lower. Taking height = 6 and order = 6 for example, the running time Adjacency List Model is around 7,000 ms, but Nested Sets Model around 9 ms.

Figure 3.8 depicts the running time, in milliseconds, of the operation implemented in MySQL stored procedure. The figure has the same exponential trend as the previous two models; however, the running time using this technique is unreasonably high. For example, with height = 4 and order = 2 (N = 15), the running time is 1831 ms (almost 2 seconds) comparing with 13 ms and 1.15 ms of

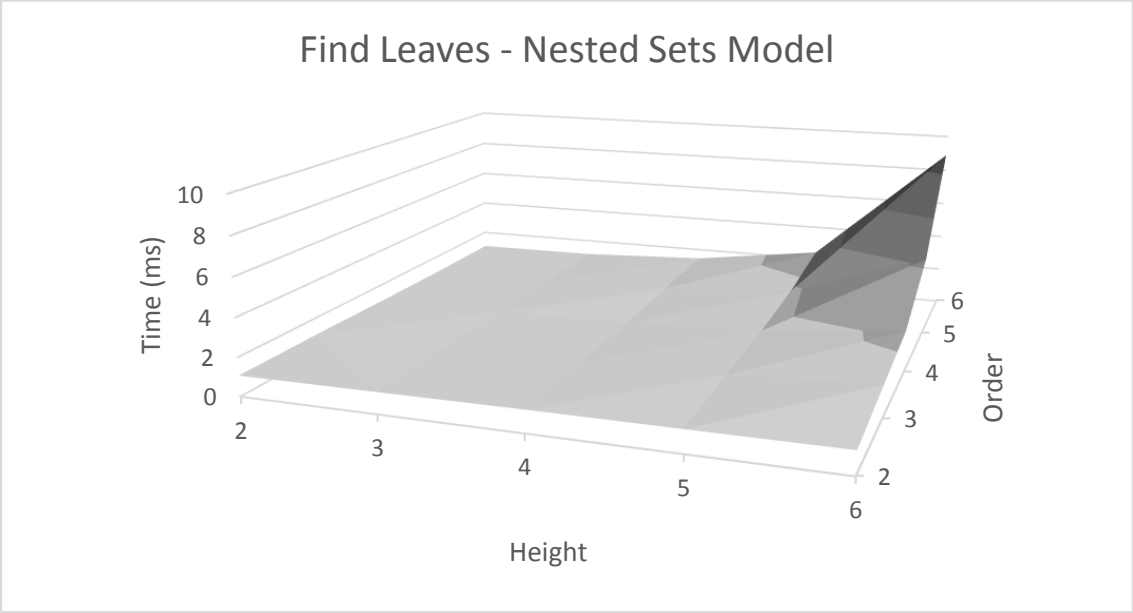


Figure 3.8: Running time of Finding Leaves using Nested Sets Model.

other two models, for very small data size. For this reason, we will no longer compare the running time of this technique to the other two.

It is interesting to reflect on this a bit. As a reminder, this technique is the Adjacency List version of the algorithm implemented in MySQL stored procedure. That means the algorithm is recursive, and we implemented a stored recursive stored procedure that is called for every node in the hierarchy. The take away here is recursive stored procedure using MySQL is possible, but it is extremely inefficient, to which we attribute this performance.

If we take a slice of Figure 3.7 and Figure 3.8 where order = 8, we can see the growth of the algorithms with respect to height, as shown in Figure 3.10. One thing to note is that the numbers in this figure for Nested Sets is scaled by a factor, due to the fact that the running time for the Nested Sets Model is so much lower than that of using Adjacency List Model. We already know that the algorithm using Nested Sets is generally better; therefore the purpose of this figure is not to compare the two approach, rather is to create a model of running time for each using the data provided.

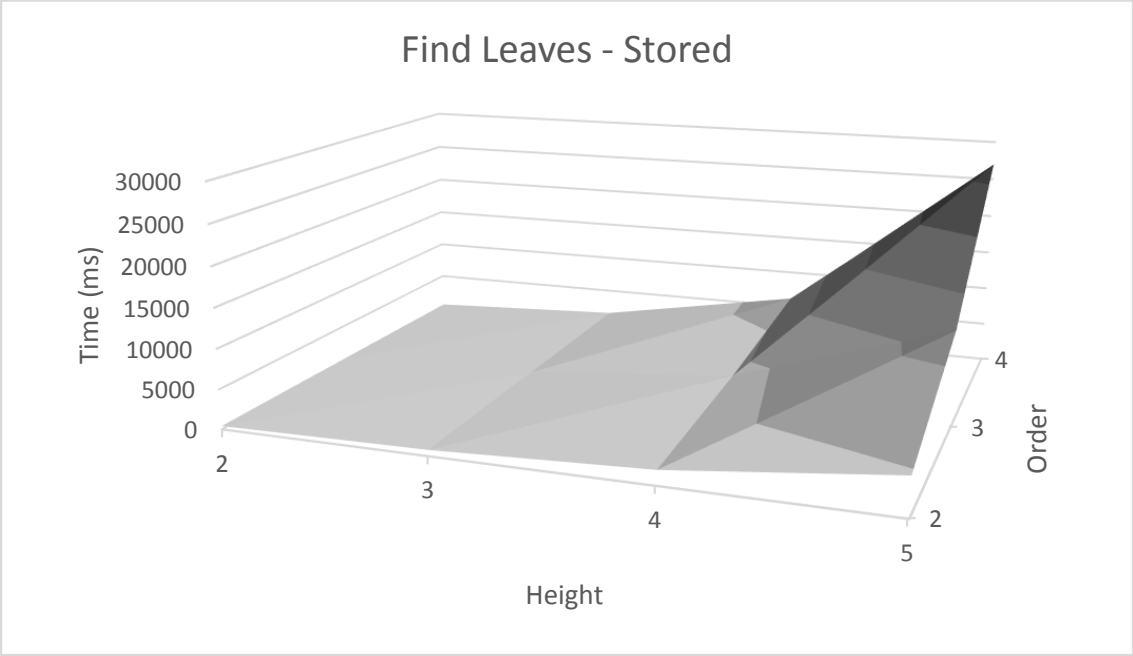


Figure 3.9: Running time of Finding Leaves using Stored Procedure.

Based on the data for height = 2 to 6 as shown in previous figure, a running time growth model was build using exponential regression as shown in Figure 3.11, where projections are shown in dotted lines. In addition, we use the data for height = 7 to validate our model. As shown in the figure, the projection from the growth model for the Adjacency List implementation fits the validation well. However, the model for the Nested Sets implementation grossly underestimate the value for the validation data. In this case, the running time jumped significantly since height = 7. This could be caused by that the experiment has not reached steady state at height = 6 and we need more data points to build a better model. Another reason could be, the database becomes extremely inefficient in processing the request for height higher than 6.

We take the same approach to analyze the running time growth rate of the operation with respect to order and the result is shown in Figure 3.12 and Figure 3.13. Once again, the number for Nested Sets Model in these figures are scaled to fit to combine the graph of both approaches in the same figure.

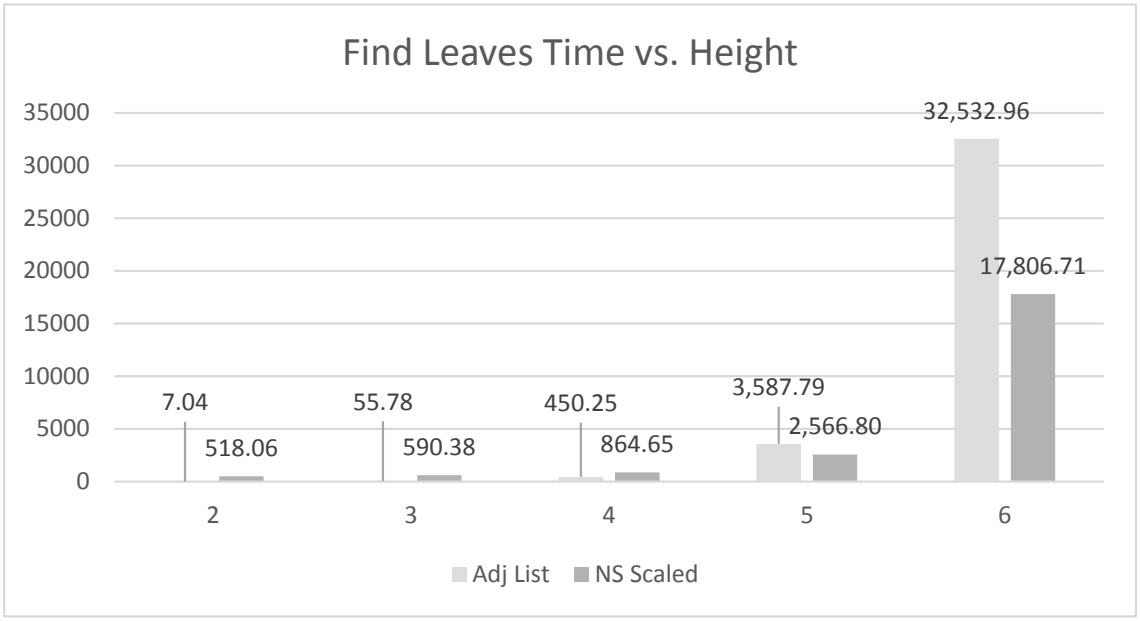


Figure 3.10: Running time with respect to height at order = 8.

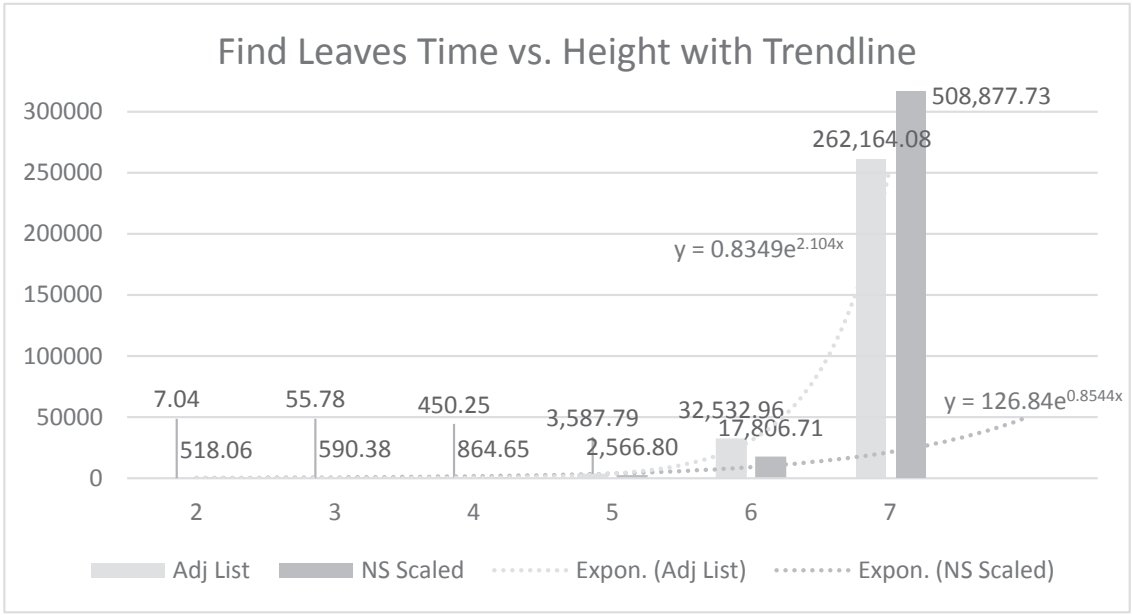


Figure 3.11: Running time with respect to height model at order = 8.

Intuitively, the running time growth rate should grow linearly with respect to the order (or width) of the tree; however, based on the initial observation of the data, it appears to be exponential, so we build a growth rate model also using exponential regression as shown in Figure 3.13 and

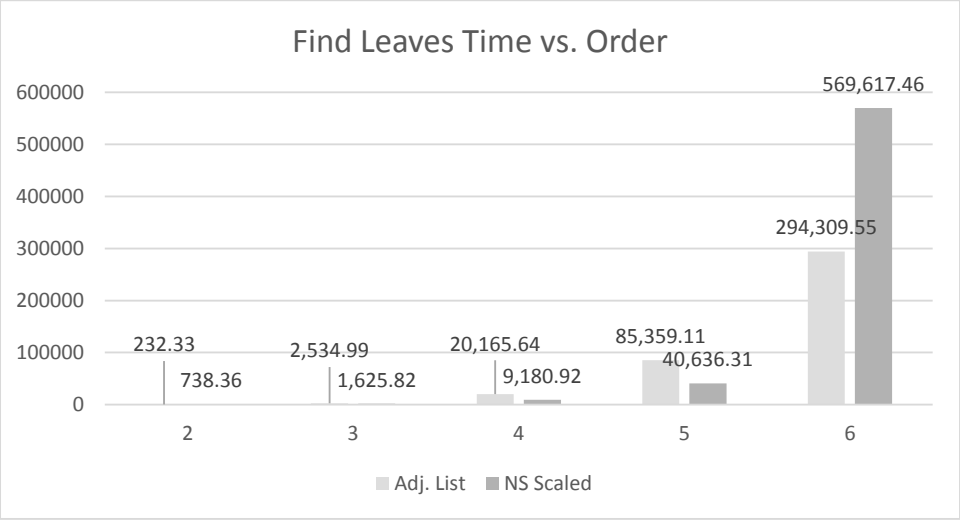


Figure 3.12: Running time with respect to order at height = 8.

using data for order = 7 as validation for our model. As one can see from the figure, the model overestimates the running time for the Adjacency List implementation. This is an indication to confirm our intuition that the growth rate is linear instead of exponential. However, the model fits the validation well for the Nested Sets implementation. This is a bit of a surprise since, this implementation usually have better performance.

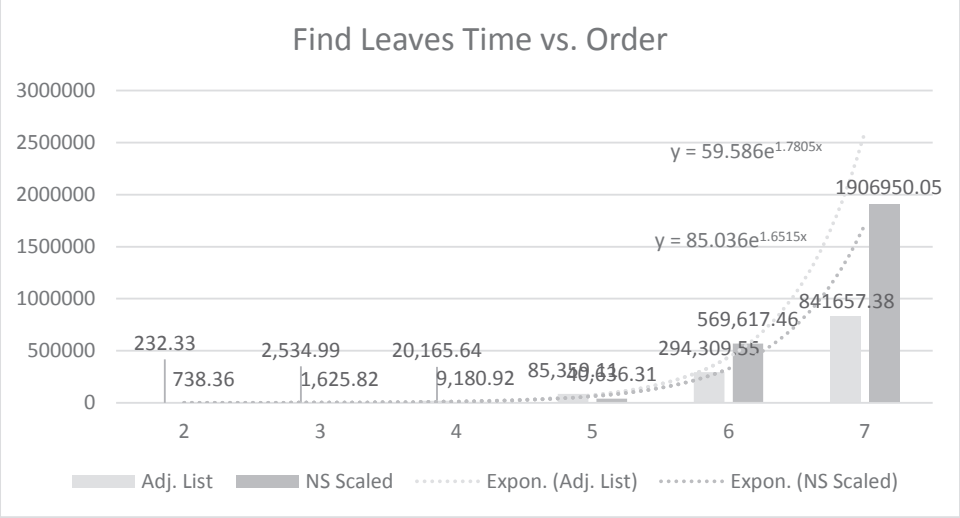


Figure 3.13: Running time with respect to order model at height = 8.

Chapter 4

Geo-Store: A Spatially-Aware SPARQL Evaluation Engine

4.1 Introduction

With the popularity of wireless networks and mobile devices, Location-Based Services (LBS) have become indispensable applications to mobile users. The global GPS navigation and LBS market size has been predicted to grow significantly from $1.6 billion in 2009 to $13.4 billion in 2014 according to IEMR's report "Global GPS Navigation and Location Based Services Forecast". In addition, the Web consists of a huge volume of data which requires the use of human intelligence to process and analyze. The main goal of the Semantic Web is to augment the Web with information so that computers can understand and exchange Web data. Consequently, it will be a trend for LBS providers to employ Semantic Web data sources to develop semantics-enabled location-based services.

The Resource Description Framework (RDF) data model is designed for interchanging schema-relaxable (or schema-less) data on the Semantic Web [94]. RDF models the linking structure of the Web as triples of the form $\langle s, p, o \rangle$, where s is a subject, p is a predicate, and o is an object. Each triple represents the relationship between the subject and the object. A collection of triples forms a directed graph, where the edges represent predicates between subjects and objects, which are represented by the graph nodes. With the increasing amount of RDF data on the Web, researchers developed specialized architectures for RDF data management named *triple stores* [1, 10, 12, 70, 97]. Generally, these solutions employ various indexing, compression, and query optimization techniques for scalable and efficient management of RDF data.

The Semantic Web is an ideal data source for supporting the state-of-the-art location-based services that employ dynamic or near real-time information. For example, a mobile user may utilize LBS to search for nearby restaurants based on recent reviews on the Web. However, to the best

of our knowledge, there are limited solutions to process spatial queries, the building blocks of most LBSs, on triple stores for supporting advanced location-based services. Therefore, the goal of the Geo-Store project is to develop novel spatial query techniques that are able to efficiently evaluate spatial queries on RDF triple stores for providing semantics-enabled location-based services. The main features of our Geo-Store system are as follows.

- The Geo-Store system employs a novel representation to model spatial features and utilizes a spatial mapping mechanism to preserve spatial locality.
- The Geo-Store system is able to effectively process both range and k Nearest Neighbor (kNN) queries, the building blocks of many LBS applications, on RDF triple stores.
- Users are able to integrate and operate the Geo-Store system on existing triple stores (e.g., RDF-3X) with limited changes.

4.2 Related Work

Location-based services are any service that takes into account the geographic location of an entity and are accessible with mobile devices through wireless networks [48]. With the prevalence of GPS-enabled mobile devices and the introduction of 4G mobile telecommunications services, various commercial LBSs, such as location-based dating, location-targeted advertisement, and child safety services, appear in our lives [3]. In addition, novel LBS applications are able to exploit online Semantic Web sources (e.g., LinkedGeoData[1]) about nearby physical entities of a user to provide personalized services [101]. Today, location-based services are applied in different fields, such as emergency response, navigation, product tracking, social networks, etc.

The Semantic Web is a group of techniques for machines to understand information and exchange knowledge on the World Wide Web. The cornerstone of the Semantic Web is a logical data model named RDF which employs triples to represent the relationships between subjects and objects. In order to efficiently manage RDF data, there are numerous systems invented for storing and

[1] http://linkedgeodata.org

querying triple collections [1, 10, 12, 70, 97]. For improving performance and scalability, Abadi et al. [1] introduced a solution by vertically partitioning the RDF data. Their solution's performance can be further improved by utilizing a column-oriented DBMS, which is a database designed specially for the vertically partitioned case. Weiss et al. [97] proposed a sextuple-indexing scheme, named Hexastore, which allows for quick and scalable general purpose query evaluation for RDF data management. Hexastore achieves significant advantages in performance compared with previous solutions for managing RDF triples. The RDF-3X engine [70] is an implementation of the SPARQL query language [95] for RDF by pursuing a simplified architecture with streamlined indexing and query processing. The design of RDF-3X completely eliminates the need for index tuning by exhaustive indexes for all permutations of subject-predicate-object triples and their binary and unary projections. However, all the aforementioned triple stores do not consider the unique features of spatial data when they encode and store RDF triples.

Recently, Perry et al. [76] presented an extension to SPARQL, named SPARQL-ST, for complex spatiotemporal queries. They implemented SPARQL-ST by extending a relational database, which is not an effective way of managing RDF triples. The Strabon system is an implementation of the data model stRDF and the query language stSPARQL [55], which are extensions of RDF and SPARQL for managing spatial and temporal data. Nevertheless, Strabon is built on top of the RDF store Sesame[2] which is not as efficient as the aforesaid new generation RDF triple stores [60]. Brodt et al. [8] proposed a solution to integrate spatial query processing into RDF triple stores. However, their design cannot efficiently evaluate queries on large-scale spatial data sets, which are essential for the state-of-the-art LBSs. Furthermore, Parliament [51, 52] is a triple store and employs a similar approach to [8] for spatial queries and storage of the data. The GeoSPARQL draft standard [15] is currently being implemented in Parliament.

[2]http://www.openrdf.org/

4.3 The Geo-Store System

Although many RDF triple stores have been proposed during the past few years, most of them were designed and optimized mainly for non-spatial Semantic Web data. In order to enable spatial query processing on RDF triple stores, the state-of-the-art method [8] is to treat all the Universal Resource Identifiers (URIs) and literals, non-spatial or spatial, equally and replace each URI and literal with an integer ID by dictionary encoding. As a result, each spatial literal (e.g., the latitude and longitude coordinates or the address) is mapped to a randomly generated ID. Afterwards, an R-tree (or its variant) [32] is created to index all the IDs referred to spatial literals. However, this dictionary encoding-based method may incur extreme inefficiency in evaluating spatial queries on RDF triple stores, especially when coping with large-scale spatial data sets [8]. In this chapter, we employ a novel representation, *GeoHilbert* – an RDF vocabulary as an extension of the Geography Markup Language (GML), to model spatial features based on their Hilbert curve [67] transformation information and to utilize Spatially Aware Mapping (SAM), instead of dictionary mapping, to encode URIs and literals for preserving spatial locality.

4.3.1 Data Representation

Because many huge data repositories published online contain geographic location or spatial relationship information, we are witnessing a new research trend of modeling geographic entities ontologically and querying their spatial relations in the Semantic Web community. As one of the Semantic Web technologies, the Resource Description Framework treats relationships as first-class citizens and, consequently, can work as an ideal tool for modeling and querying complex and large amounts of relations between spatial entities. The XML formatted semantic data can be converted to the corresponding RDF representation with minor modifications. For example, Listing 4.1 shows an XML data excerpt from the OpenStreetMap project[3]. This snippet describes the referred information (semantic and spatial) about a restaurant in Pasadena, California, USA. Its corresponding RDF representation is demonstrated in Listing 4.2. Specifically, *gml* represents

[3]http://www.openstreetmap.org/

the namespace[4] of the GML standard, and *gstore* corresponds to the namespace[5] of our Geo-Store system.

Listing 4.1: An XML data excerpt from the OpenStreetMap project.

```
1 <node id="738330640" lat="34.1135498" lon="-118.1235345" user
    ="AM909" uid="82317" visible="true" version="1" changeset
    ="4737707" timestamp="2010-05-18T11:26:40Z">
2   <tag k="amenity" v="restaurant" />
3   <tag k="cuisine" v="american" />
4   <tag k="name" v="Colonial Kitchen" />
5   <tag k="source" v="usgs_imagery;survey;image" />
6   <tag k="source_ref" v="AM909_DSCU3253" />
7 </node>
```

Listing 4.2: The RDF representation of the restaurant described in Listing 4.1.

```
1 ...
2 gstore:Point738330640 gml:amenity "restaurant"
3 gstore:Point738330640 gml:cuisine "american"
4 gstore:Point738330640 gml:name    "Colonial Kitchen"
5 gstore:Point738330640 gml:pos     "34.1135498 -118.1235345"
6 ...
```

4.3.2 System Architecture

Figure 4.1 shows the system framework of the Geo-Store system, which consists of four main components: query parser and planner module, spatially aware mapping module, internal processing module, and dictionary decoding module.

[4]http://www.opengis.net/gml
[5]http://example.org/gstore

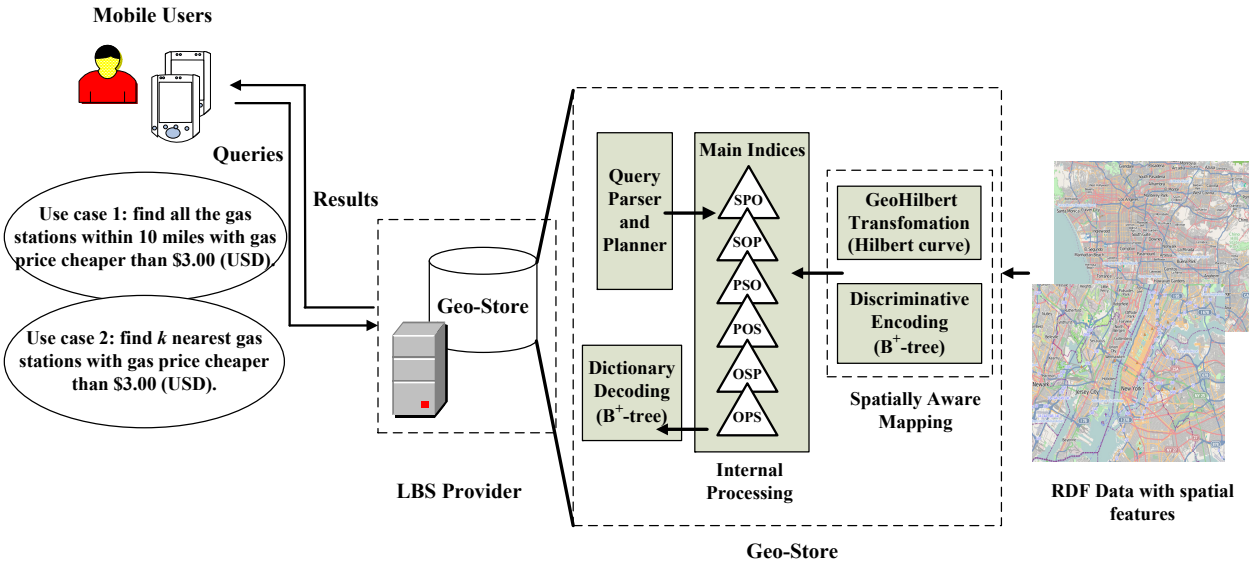


Figure 4.1: Geo-Store system architecture and use cases.

Spatially Aware Mapping Module

In most of the existing triple stores, each URI or literal is replaced with a unique integer ID by dictionary mapping because the triples may contain very long URIs and string literals. In our system, in order to support efficient spatial query evaluation on triples, we extend the standard dictionary encoding and design a Spatially Aware Mapping (SAM) approach to encode all the URIs and literals, as well as preserve spatial locality. SAM includes two major components, *GeoHilbert Transformation* and *Discriminative Encoding*. With GeoHilbert Transformation, original RDF triples are transformed to the corresponding GeoHilbert representation. In addition, SAM assigns an ID to each URI and literal in a discriminative way in order to maintain spatial locality by executing the discriminative encoding component.

GeoHilbert Transformation We utilize a novel RDF representation, GeoHilbert, to model spatial data in Geo-Store. GeoHilbert is designed for incorporating the Hilbert curve-based spatial transformation information into original RDF data. GeoHilbert consists of a subject, *HilbertMapping* and four predicates: *StartPoint*, *Order*, *Orientation*, and *Pos*. Specifically, the values of *StartPoint*, *Order*, *Orientation*, and *Pos* are the location of the Hilbert curve starting point, the

curve order O, the curve orientation θ, and a Hilbert value translated from a given spatial location, respectively.

Listing 4.3 demonstrates the corresponding RDF data in the GeoHilbert representation for the restaurant described in Listing 4.1. As shown in Listings 4.2 and 4.3, the GeoHilbert representation appends to the original RDF representation an additional RDF statement, {*gstore:Point738330640*, *gstore:Pos*, *"208083"*}, which stores the relative position of the referred point along the Hilbert space-filling curve.

Listing 4.3: The GeoHilbert representation of the restaurant described in Listing 4.1.

```
1  ...
2  gstore:HilbertMapping gstore:StartPoint  "32.3372200
     -114.1369000"
3  gstore:HilbertMapping gstore:Order       "10"
4  gstore:HilbertMapping gstore:Orientation "Up Left"
5  ...
6  gstore:Point738330640 gml:amenity "restaurant"
7  gstore:Point738330640 gml:cuisine "american"
8  gstore:Point738330640 gml:name    "Colonial Kitchen"
9  gstore:Point738330640 gml:pos     "34.1135498  -118.1235345"
10 gstore:Point738330640 gstore:Pos  "208083"
11 ...
```

Discriminative Encoding The Discriminative Encoding (DE) component takes the GeoHilbert representation based data as the input. For each literal following the predicate *Pos*, DE assigns it a positive ID which is exactly the same as its Hilbert value. For example, in Listing 4.3, the literal *"208083"* in the triple {*gstore:Point738330640*, *gstore:Pos*, *"208083"*} will be assigned with ID *208083*. For all the other literals or URIs, DE acquires a random integer by calling a dictionary encoding function, and then returns the opposite of that integer as the ID (e.g., a randomly generated

integer "1000" will be returned as "-1000"). All the assigned IDs are maintained in a B^+-tree to speed up the dictionary lookup.

Internal Processing Module

Generally, the evaluation of SPARQL queries is based on pattern matching. In this system, we maintain in memory all six possible permutations of *subject* (S), *predicate* (P), and *object* (O) in six separate indices in order to guarantee that every possible query pattern in a SPARQL query with variables in any position of a triple can be answered by only performing a single index scan. Specifically, these permutations are named as SPO, SOP, OSP, OPS, PSO, and POS indices, respectively. Notice that instead of the original URIs or literals, all the six indices here consist of integer IDs, which are assigned by the discriminative encoding component. This ID-based indexing scheme can both save memory space and accelerate join processing.

Query Parser and Planner Module

When Geo-Store receives a SPARQL query q, it parses q, identifies the IDs that have been assigned to the literals in q according to SAM, and employs the retrieved IDs to replace the corresponding literals in q. By executing these processes, the subsequent evaluation of q purely relies on the comparisons of IDs instead of the original literals. A query evaluation plan can then be generated accordingly.

Dictionary Decoding Module

After the SPARQL query is evaluated by the internal processing module, the dictionary decoding module transforms the resulting IDs back to their original literals as the query results. In Geo-Store, we employ a B^+-tree structure to implement this ID-to-literal mapping.

4.4 Semantics Enabled Location-based Services with Spatial Encoding

The SPARQL query language is standardized by the World Wide Web Consortium (W3C) for querying RDF data. Evaluation of SPARQL queries is based on pattern matching on the target RDF graph. A pattern may contain variables that are bound to a URI or a literal. In addition, the SPARQL query language provides a number of built-in FILTER functions, which can be easily extended to support spatial operations (or constraints). Because range and k nearest neighbor queries are the building blocks of location-based services, we elaborate on how to evaluate the two important query types by utilizing our system in this section.

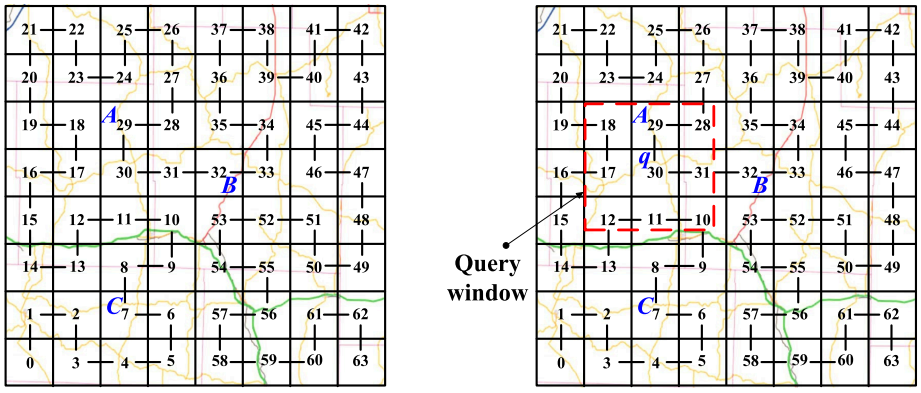
4.4.1 Range Queries

Listing 4.4 shows a sample range query which is launched to retrieve the latitude and longitude values of all the qualified gas stations with the two selection conditions: (1) the location is less than 10 miles away from the query point (the mobile user's current location), and (2) the gas price is cheaper than $3.00 per gallon.

Listing 4.4: The sample range query in Geo-Store (use case 1).

```
1 SELECT ?coordinates
2 WHERE {
3   ?point a "Gas Station" .
4   ?point gml:pos ?coordinates .
5   FILTER (gstore:range(?point, CURRENT_LOCATION, 10, "mile"))
6   ?point gstore:gaspriceUSD ?price .
7   FILTER (?price < 3.00)
8 }
```

In Geo-Store, each spatial object is annotated with its Hilbert value information based on the GeoHilbert representation. Figure 4.2(a) illustrates an example of mapping spatial objects in a two dimensional space into their Hilbert values. In Figure 4.2(a), the entire space is divided by the



(a) Hilbert curve transformation. (b) Range query Q_R.

Figure 4.2: Hilbert curve transformation and range query.

Hilbert curve into 64 grids with their unique Hilbert values, and we can acquire the Hilbert values of the spatial objects *A*, *B*, and *C*, as 29, 32, and 7, respectively. Depending on the desired resolution, more fine-grained curves can be recursively generated based on the Hilbert curve generation algorithm. Figure 4.2(b) demonstrates how a range query can be processed in our system by taking advantage of the Hilbert values of spatial objects. As depicted in Figure 4.2(b), the query point is q and the query window of the range query Q_R, highlighted in red, covers the three Hilbert curve segments [10-12], [17-18], and [28-31]. After obtaining the above three curve segments, our system retrieves all the spatial objects whose Hilbert values are embraced by the three curve sections and treats the retrieved spatial object set $\mathbb{R}'$ as the inclusive query result. Subsequently, our system examines all the spatial objects in the grids that *partially* overlap with the query window (i.e., grids [10-12], [28] and [31]) to check if their exact locations (i.e., latitude and longitude values) are within the query window by dictionary lookup. Finally, the exact query result $\mathbb{R}$ is returned to the user after filtering out those objects whose locations are outside the query window in the partially overlapping grids.

4.4.2 k Nearest Neighbor Queries

In this subsection, we extend our range query solution to evaluate k nearest neighbor queries efficiently in Geo-Store. Listing 4.5 demonstrates a sample k nearest neighbor query which is

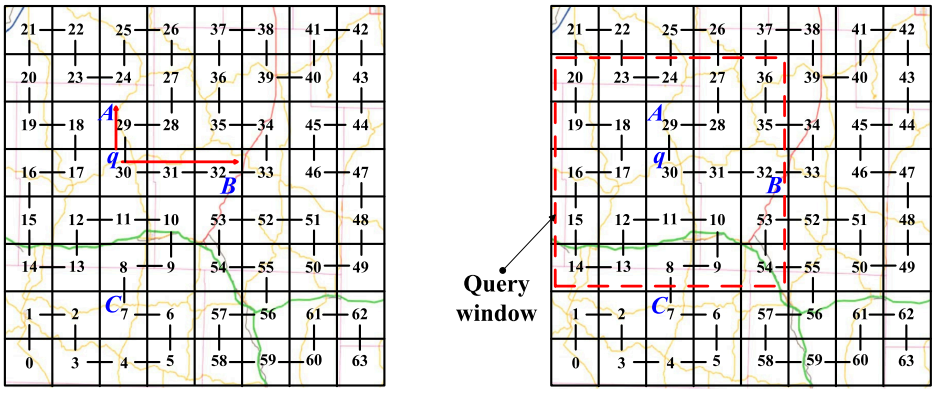
issued to retrieve the latitude and longitude values of the gas stations that are among the k closest gas stations to the query point with a listed gas price cheaper than $3.00 per gallon.

Listing 4.5: The sample k nearest neighbor query in Geo-Store (use case 2).

```
1 SELECT ?coordinates
2 WHERE {
3   ?point a "Gas Station" .
4   ?point gml:pos ?coordinates .
5   FILTER (gstore:NN(?point, CURRENT_LOCATION, k))
6   ?point gstore:gaspriceUSD ?price .
7   FILTER (?price < 3.00)
8 }
```

Given a query point q, Geo-Store searches the spatial objects in both the ascending and descending directions of Hilbert values until k spatial objects are found, and then records the result set as $\mathbb{S}$. Supposing the object o has the longest distance to q in $\mathbb{S}$, $Distance(q, o)$ (the distance between q and o) is set as the *search upper bound* for the subsequent range query. Afterwards, Geo-Store launches a range query Q_R with $Distance(q, o)$ to decide the query window size and then acquires the query result $\mathbb{R}'$. Next, Geo-Store identifies the top k objects in $\mathbb{R}'$ based on their respective distances to q in order to derive the final query result $\mathbb{R}$.

Figure 4.3 demonstrates the evaluation of a k nearest neighbor query with Geo-Store. As shown in Figure 4.3(a), based on the query point q, Geo-Store first searches for spatial objects with Hilbert values ≥ 30 and < 30 in parallel until k objects are discovered. Next, assume that the spatial object that has the longest distance to q among the above k objects is object B. Then, a range query Q_R with the distance between q and B as the search upper bound is issued; Q_R returns the result set $\mathbb{R}'$, as depicted in Figure 4.3(b). In this example, set $\mathbb{R}'$ encompasses objects which fall on the five Hilbert curve segments, [8-20], [23-24], [27-32], [35-36], and [53-54]. Finally, Geo-Store computes the top k objects in $\mathbb{R}'$ with the shortest distance to q as the exact query result.



(a) Finding the search upper bound.

(b) The derived range query Q_R.

Figure 4.3: k nearest neighbor query example.

4.5 Experimental Validation

For performance evaluation, we compared our Geo-Store system with Brodt's system [8] and the Strabon system [60] by measuring their response time on processing two popular location-based spatial query types, range and kNN. We acquired the locations of points of interest (POI) in California from the U.S. Geological Survey[6]. This real-world data set contains one million POIs distributed over the state of California with 63 different POI categories, including airport, hospital, school, populated place, road junction, etc. Each category exhibits a distinct density and distribution. For each result in this section, we ran 500 corresponding queries with distinct query points whose locations were randomly generated within the state of California. All the experiments were conducted on the same Windows machine.

4.5.1 Efficiency Comparison

In this subsection, we focus on comparing the efficiency of Geo-Store, Brodt's, and Strabon in terms of query response time.

[6]http://cumulus.cr.usgs.gov/

Range query

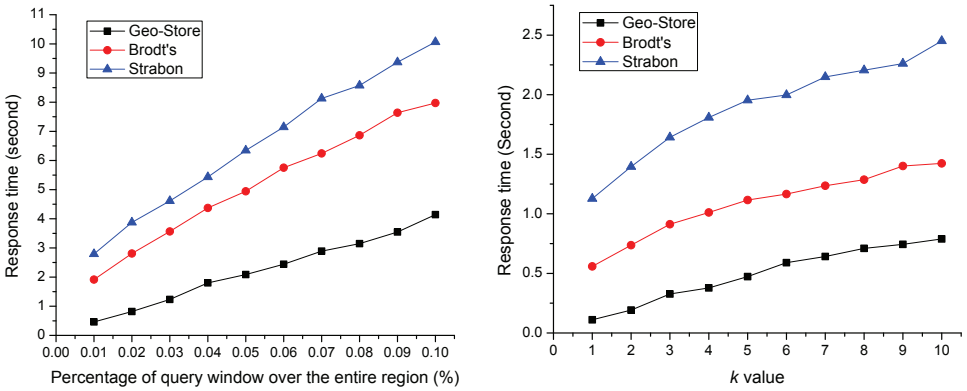
We first report our experimental results of range queries. We gradually increased the area of the query window to investigate its impact on query response time. Here we specify the area of a query window by using the percentage of the region of California. For example, a query window with the percentage of 0.01% represents an area of around 16 square miles (given the region of California is around 160,000 square miles). As shown in Figure 4.4 (a), with the enlargement of the query window, the response time kept increasing for all systems. However, our Geo-Store system always outperformed Brodt's and Strabon. For instance, with the query window of 0.01%, Geo-Store only required 0.422 seconds on average to execute the range query, while Brodt's and Strabon needed 1.916 seconds and 2.792 seconds, respectively. The advantage of Geo-Store over the other two systems, in terms of efficiency in evaluating range queries, can be explained as follows.

By utilizing the Hilbert curve based transformation, Geo-Store manages to maintain a roughly one-to-one mapping between Hilbert values and POIs. Therefore, in most cases, the spatial relation between two entities can be determined by comparing their respective Hilbert values. As a result, if a POI is identified as beyond the query window by the examination of its Hilbert value, it can be filtered out at an earlier stage, and there will be no need to check its exact latitude/longitude information on disk. On the contrary, in Brodt's and Strabon (both employ the R-tree index), each Minimum Bounding Rectangle (MBR) contains numerous POIs (i.e., usually more than half of the fan-out value), resulting in a much larger amount of I/Os to retrieve the latitude/longitude values on disk in order to decide if a particular POI satisfies a spatial selection operation or not.

k nearest neighbor query

Next we study the efficiency of all three systems in evaluating k nearest neighbor queries. We varied the k value from 1 to 10. Figure 4.4 (b) shows that when we elevated the k value, all the systems required a longer response time to identify the k closest neighbors. Nevertheless, as demonstrated in Figure 4.4 (b), the response time needed by Geo-Store was significantly reduced,

compared to the other two systems. For example, with k equal to 3, Geo-Store only required 0.325 seconds on average, while Brodt's and Strabon needed 0.913 seconds and 1.642 seconds, respectively. Geo-Store demonstrates a much higher efficiency than the other two systems in processing k nearest neighbor queries because Hilbert values employed in Geo-Store can provide a more precise location estimation of each POI than MBRs used in R-trees, which improves query evaluation performance.



(a) Range query. (b) k nearest neighbor query.

Figure 4.4: Performance comparison of the three systems.

4.5.2 Integration with Existing Triple Stores

As discussed in Section 4.3, one of the clear advantages that Geo-Store provides over Brodt's solution is that no complicated spatial index, such as an R-tree, is needed to be maintained in Geo-Store. The indexing mechanism in Geo-Store is completely consistent to the six separate indices design [97], which is employed in most existing triple stores [1, 10, 70, 97] for RDF query evaluation. On the other hand, in [8], a separate R-tree is responsible for indexing all the spatial IDs while all the IDs, spatial or non-spatial, are indexed by a B^+-tree. Therefore, by using our design, current RDF triple stores can be easily extended to support location-based services with limited cost on integration.

Chapter 5

Conclusion and Future Work

5.1 Efficient Evaluation of Skyline Queries in Wireless Data Broadcast Environments

In chapter 2, we presented a broadcast data stream allocation technique (TDI) that utilizes the R-Tree and performs a depth-first traversal of the index to create a distributed index. The goals of TDI are to facilitate query processing from broadcast data, to reduce the index overhead (IP) and to improve the initial index probe. TDI is able to achieve all these with reasonable efficiency. TDI is a flexible R-tree based index and supports skyline queries as well as other data query types. The allocation distributes b^h number of index segments among the broadcast program to reduce the initial index, and at the same time keeps the index overhead low. The simulation results for index percentage show that TDI performs very well with two levels of replication and follow the efficiency of one-time index with only a 16% increase in index overhead.

In addition, we introduced record-based pruning skyline (RPS) and index-based pruning skyline (IPS) algorithms. The experiments show that both algorithms are capable of evaluating skyline queries of combined min and max attributes with reasonable tuning time and dominance tests. The index-based skyline has always performed better than the point-based skyline, in some cases by several factors. The performance of the algorithms is also affected by the data arrangement and R-Tree implementation. From our simulation, we find that R-trees that index records with lower attributes first perform better for min skyline queries, and vice versa.

The simulation results show that the approach also performs well with data of higher dimensions. The index overhead decreases as the number of records remain constant and the number of data dimensions increase. This is due to the fact that the growth of dimensionality does not make the index tree grow "taller" and does not incur the cost of new nodes when the index grows.

The height of an index tree does increase as the number of records increase, but as seen in Figure 2.14(a), the growth of the index is not as fast as the growth of the amount of data; therefore, the index overhead decreases as records increase.

For the future work, we would like to expand our study of the broadcast aspect of this work. We have done preliminary design of the broadcast data serialization technique and performed analysis and experiments using simulations; in order to have an in-depth study of this aspect, experiments with physical wireless devices must be conducted.

In addition, an open issue of our skyline algorithm is how to convert a text-based attribute into an numeric value that can be indexed an used by our algorithm. An simple answer is to use hash function, however, we must consider the natural order of text strings and the mean of a string is greater than or less than the other.

5.2 Hierarchical Data in Relational Database Management Systems

In chapter 3 we conducted performance evaluation of three implementations of hierarchy in MySQL relational database. The first is the Adjacency List Model, which is realized by storing all nodes of the hierarchy in a single Nodes table, and each node, represented by a record in the table, contains a self-referencing foreign key that references the parent node.

The second is the Nested Sets Model, which is an attempt to improve the performance of data retrieving operations over the Adjacency List Model. This model is an extension of the first in that each node (or record) contain two additional *left* and *right* fields that defines the subset relationship. Left and right fields are then indexed using B-Tree for fast access. With this extension, one can easily check if a node is in the hierarchy of another node.

The third implementation is the Stored Adjacency List, in which the same algorithm used in Adjacency List Model implemented in MySQL Stored Procedure. This technique is used to verify if stored procedure is a viable way for performance tuning for hierarchy operations.

Performance evaluation were conducted using the operations defined in Section 3.2.3. Operations are divided into three performance groups: O(log N), O(N), and O(1). Members of the same

group have similar performance characteristic; therefore, one operation from O(log N), and O(N) is chosen for performance evaluation as representative for the group. The evaluation for group O(1) was conducted, but not reported in this chapter, because the run time is negligible and not influenced by input size. The operation to find the root of a node was chosen to represent O(log N) group, and the operation to find all leaf nodes of a node was chosen to represent O(N) group. The performance of the models are measured in running time of the operations, in which the lower the running time the better.

For group O(log N), the evaluation reported in Section 3.5.4 shows the expected running time performance. For the Adjacency List Model, the growth rate of the algorithm is linear with respect to the height and constant with respect to the width (order) of the hierarchy. The Nested Sets Model is a huge improvement over the first in this evaluation. Based on the evaluation, the running time growth rate is constant for both dimensions. For the stored implementation, the running time is better than both other approaches. However, the project model is slow-growing linear with respect to height, and constant with respect to order.

In Group O(N), the Nested Sets Model still significantly outperformed Adjacency List Model; however the running time of the stored approach is unreasonable high for very small data sets. For the Adjacency List Model, the running time, as expected, have exponential running time growth with respect to the height of the tree. From the evaluation result, the growth rate with respect to width also appears to be exponential; however, our validation does not fit the exponential model, therefore we believe the growth is a steep linear model.

For the Nested Sets model, the growth rate is also exponential. Although the running times are much lower than that of the other approach, the growth is faster. Frankly, we did not expect an exponential growth rate due to the optimizations that has been done. We believe this could be caused by our implementation. As for the time complexity against the width, it is also exponential. This is also unexpected. Again this is due to that the implementation requires table joins, which degrades the performance very quickly as the data size increases.

An open issue and future work of our index technique is updating an hierarchy structure. A scheme must be developed so that the left and right values are assigned in the way that inserting or removing a node from the hierarchy is efficient and does not cause reassignment of the left and right values of all the nodes in the hierarchy. A possible approach is allocate the left and right values in the way that it leaves enough space for future records to be inserted.

5.3 Geo-Store: A Spatially-Aware SPARQL Evaluation Engine

The increasing amount of RDF data containing location-based information calls for the development of systems which support effective evaluation of location-based services on RDF triple stores. In our Geo-Store project, we implement a system which is capable of querying heterogenous data sources and providing semantics-enabled location-based services with high efficiency. The Geo-Store system confers the following advantages. First, Geo-Store utilizes Spatially Aware Mapping to preserve spatial locality during encoding. Second, as our experiments demonstrate, Geo-Store allows for effective processing of range and kNN queries. Third, existing triple stores can be easily integrated with Geo-Store with limited integration cost.

For future work, we plan to extend Geo-Store to support other novel RDF query languages specifically designed for spatial data management, such as GeoSPARQL and stSPARQL, by expanding the query parser module and related components. In addition, we will support more spatial query types, such as spatial join, in the next phase of this project.

Appendix A

Computerized Control and Data Acquisition System for MDPX

A.1 Introduction

Plasma is a state of matter in which enough energy is obtained from an outside source for the electrons to become unbound from the nucleus of individual atoms. By definition a plasma is ionized gas which consists of electrons and positively charged ions. Due to the free electrons, plasma, unlike gas, is an excellent conductor of electricity.

Plasma is rarely found without some kind of dust particulates. When Lewi Tonks and Irving Langmuire conceived the term plasma in 1929, they reported seeing "globules" in the gas discharge that can be tracked with naked eyes [89, 92]. A loose definition of dusty plasma is an ionized gas with micron-sized particulate components. In a plasma, the dust particulates become electro-statically charged by surrounding plasma. The interaction of these charged macro-particles increase the complexity of the state of the plasma environment; therefore, dusty plasma is also called complex plasma.

We can more strictly define the property of a dusty plasma. Given the radius of the dust particulates, r_d, the average distance between dust grains, a, and the Debye radius of the plasma, λ_D, the situation $r_d \ll \lambda_D < a$, where dust grain is sparse, is referred to as 'dust in plasma'; while the situation $r_d \ll a < \lambda_D$, where dust grains interact with each other in a collective behavior, is referred to as 'dusty plasma' [82].

In addition to dust particulates, magnetic fields are also a common component of a dusty plasma system. Internally, the plasma and charged particulates generate magnetic fields, when ions and charged particulates are in motion. However, due to the low mass of the dust particulate, the internal generate magnetic fields are small. More often, magnetic fields are introduced from external sources. In planetary scale, a magnetic field could come from nearby star or planet. In

laboratories, strong magnetic fields are often generate to modify the behavior of a dusty plasma system.

Without a magnetic field, the predominate force in the system is the electrostatic force of the charged particulates. In a laboratory setting, gravity also influences the interaction of the particulates. In order for the charge particulate to escape the influence of gravity, the electrostatic forces of the particulates must be overcome the force of gravity. Due to that the dusty plasma is composed of ions and charge particulates, an external magnetic field could have predominant influence to the behavior of the system.

The study of magnetized dusty plasma can answer fundamental questions of astrophysics. Studies have found that over 99% of the matter is plasma with dust being an omnipresent ingredient [82]. Magnetized dusty plasma can also be found in the tail of comet [71] and according to the data received from the Voyager I spacecraft, the interstellar medium that surrounds our solar system is composed of ionized gas and dust [9, 16, 98]. Studies done in this area has the potential to give us a deeper insight on the the intricate process of star formation [66], the formation of early planets from charged dusty plasmas, and the interaction between ice particulates that formed the pattern of the rings around Saturn [27, 29].

The study of magnetized dusty plasma also has scientific and industrial applications. In integrated circuit manufacturing process, dusty particulates are found in plasma etching and ashing process. A precise control of the plasma is required to reduce error rate and increase the yield of usable silicon chips. Magnetic confinement fusion techniques also utilize magnetic fields to control hot fuel in form of plasma.

The Magnetized Dusty Plasma Experiment (MDPX) is a new research facility at Auburn University started in late Spring of 2014 to study magnetized dusty plasma under a set parameters that has not been done previously. The goal of the facility is to support new types of experiments with emphasis on the following characteristics:

1. The study of plasma under high uniform and non-uniform magnetic fields to study the effect of charge on dust particulates.

2. To sustain highly magnetized dusty plasma for a extended period of time to study the behavior of the system in steady-state.

The core of MDPX research facility is the newly designed magnetized dusty plasma device. The design of the research plasma device is a multi-institutional collaboration between Auburn University, University of Iowa, and University of California in San Diego. The preliminary design and mission was described by Thomas, et al. with incorporation of lessons learned from earlier fusion and plasma experiments. The goal of the device is to be able to generate magnetized plasma of uniform magnetic field up to 4 Tesla and non-uniform magnetic fields with gradients of 1 to 2 Tesla per meter with a flexible, removable vacuum chambers that provides substantial probe and optical access to the plasma [90].

The focus of this chapter is to outline the computerized control and data management system of the MDPX project. This chapter is organized as follows. Section A.2 will cover related works, which contains a survey of previous experiment control and data management systems. Of particular interests are EPICS [47] and MDSPlus [85], which is covered in the second part of the related works section. Section A.3 describes the machines configuration of MDPX, and major components of the device. Since the focus of this chapter is the control software of the system, this section will be brief. Section A.4 describes the control and data management software of the system and is deviced into three components: LabVIEW Control software, searchable relational database, and web data access portal. This section contains detailed design of the relational database. Finally, Section A.5 concludes this chapter.

A.2 Related Works

In this section we cover previous related plasma fusion experiments and their control and data acquisition systems. One must note that most fusion experiments are complex systems consists of many components and parts. For example, the ITER [77] system consists of the vacuum vessel component, magnet system, cryostat component, and cooling systems; each of the components could have its own computerized control and safety system. In this section, we will not go into

detail of the each component of the system. Instead, we will only summarize the computerized control system and data acquisition and management system of each project.

In this section, we focus on three components of the control system. The first is the hardware and communication protocol that is used to connect the control system to the fusion experiment device. The second is the software component that is used to control experiments and acquire experimental research data. And the third is the data management software and technique.

A.2.1 Plasma Fusion Experiments

Tokamak Fusion Test Reactor (TFTR)

One of the earliest fusion experiment is the Tokamak Fusion Test Reactor (TFTR) that was built at Princeton Plasma Physics Laboratory (PPPL) in Princeton, New Jersey, USA. During its 15 years of operation from 1982 to 1997, the experiment achieved a few scientific advancements. In 1993, it was world's first magnetic fusion device to perform extensive experiments with plasma composed of 50/50 deuterium and tritium [86]. In 1994, it produced a then world-record 10.7 megawatts of fusion power, and in 1995, attained record temperature of 510 million °C, a temperature required to produce fusion power from the plasma [49].

The Central Instrumentation, Control and Data Acquisition (CICADA) system provides centralized service for the coordinated operation of the TFTR facility [79]. Process control is performed on Gould SEL 32/87 computers which is interfaced to the experiment device through the CAMAC hardware interface. In the same manner, the diagnostic hardware is interfaced through the CAMAC links on the diagnostic subsystem computers.

A specialized software event control system build by software engineers coordinates the events for a particular process. This component is responsible for synchronization within the CICADA software system.

The experiment device currently produce 6 megabytes of data per shot. The system performs data reduction, display, and archiving between shots. The data are stored on disk as normal files

on the file system, without a special database management software. Additional Data Analysis/Reduction Time Sharing (DARTS) component performs off-line analysis of the data.

Joint European Torus (JET)

The Joint European Torus (JET) [22, 25, 78] (1994–present) is one of the largest Tokamak fusion experiment device in the world. The facility is part of the Culham Centre for Fusion Energy (CCFE) located in Culham, Oxfordshire, in England, UK. Its initial plasma experiment was ran in 1983. Due to its flexible design, the device was upgraded and reconfigured several times to keep pace with scientific research and advancements. Part of upgrades is to use JET as testbed for future devices such as ITER [19, 20] and DEMO [53], which are based on the design of JET.

The first version of computerized control system for JET is named Control and Data Acquisition System (CODAS) [93]. The system consisted of computer consoles controlling different part of the system interfaced with CAMAC. Software engineers created specialized software for the project using combination of ANSI FORTRAN[1] and other high-level interpreted languages.

In 1990, the computer hardware and operating system was upgraded. The chosen architecture were SPARC computers with Systems V, SOLARIS-2, and SUNOS-4 operating systems. FORTRAN code was also moved to C language [56].

In 1994, the data acquisition is upgraded to be called Fast Central Acquisition and Trigger System (CATS) [5]. The reason for the upgrade is to accommodate diagnostic components producing data at very high rate (250KHz-1MHz). In addition, the system was designed to reduce incoming data and to automatically detect plasma phenomena that are interesting to researchers. The system was designed with a network of 40Mhz Texas Instruments TMS320C40 (C40) parallel digital signal processors and HELIOS parallel operating system. As far as I can tell, there is still no specific database software for data storage. Long-term data are stored on disk. In 2008, a web-based interface was proposed to the system [38].

[1] http://en.wikipedia.org/wiki/Fortran

Alcator C-Mod

Alcator C-Mod (1991–present) is a tokamak research fusion device located at MIT Plasma Science and Fusion Center (PSFC) in Cambridge, Massachusetts, USA. Preceded by Alcator A and Alcator B experiments, Alcator C-Mod is the tokamak device that has achieved the highest magnetic field (3-9 T toroidal) and plasma pressure in the world. It is one of the major fusion research facility in the United States.

The first version of the control system for Alcator C-Mod consists of DEC AlphaStations, VAXstations running UNIX-based OpenVMS operating system. The argument for using OpenVMS the built-in networking capability its stability. The data acquisition system was controlled by a VAXserver connected to CAMAC interfaces [23]. This system is called a hybrid system because it combines an analog signal path with digitally controlled gains, utilizing digital-to-analog converters (DAC) [39].

In 2005, the control system was converted from a hybrid system to an all digital processor system called digital plasma control system (DPCS) [84]. The new system utilizes two 64 input, 16 output CompactPCI digitizers attached to a rack mounted Intel Xeon Server running Red Hat Linux. The reason for the upgrade was that older hardware standards and data communication protocols were becoming obsolete.

The software component of both old and new hardware configuration were written in IDL, an interpreted language create by Research Systems Inc. One of the major software create for running experiments is called PCS, which is used to design experiment pulses either offline or between shots.

The system utilizes an hierarchical data management software called MDSPlus [85]. MDSPlus is used for storing and loading parameters into experiment hardware during the initialization as well as storing research data produced from pulses.

DIII-D

DIII-D is a tokamak fusion device constructed in 1980s by General Atomics in San Diego, California, USA. The project pioneered technological advancements in fusion research and is one of the largest tokamak device in the world, after JET in UK and JT-60U in Japan.

The computer control of multipulse Thomson scattering diagnostic system [28] consists of Motorola 68030 microprocessors running VMEexec real-time operating system. A MicroVAX 3400 computer is used for tasks not requiring real-time processing such as initialization and post-shot analysis. Computers are connected via MicroDNET interface.

Data acquisition hardware consists of a set of CAMAC crates located on a MicroVAX-based serial highway. Data are read using standard CAMAC calls. Polychromator data are acquired by LeCroy FERA (fast encoding and readout ADC) systems. Lasers are precisely controlled by the real-time multiprocessing computers.

Large Helical Device (LHD)

Large Helical Device [42, 43] (1998–present) is a superconducting helical stellarator fusion research device built by the National Institute for Fusion Science (NIFS) in Toki, Gifu, Japan. LHD is one of the largest stellarator in the world with major radius of 3.9 meters, minor radius of 0.65 meters and magnetic field of 3 to 4 Tesla [43].

The computerized control and data acquisition system of LHD is named LABCOM. In the initial configuration LABCOM [69], which was designed to handle short-pulse experiments ($\sim$10 seconds), the data acquisition system was composed of around 30 IBM-compatible PCs running Windows NT operating system, CAMAC digitizer modules, and SCSI optical extender for data transfer. In addition, VME computers running real-time Tornado/VxWorks operating system were used for diagnostics and real-time control components.

The Windows NT computers controls the data acquisition through CAMAC interface. The control software and CAMAC driver was written in C++. The system utilized O2 object database system (ODBMS). The rationale for using this database system is that, objects used in C++ can

be directly saved into the database. The system contains two tiers storage. One is a local 50 GB RAID storage which is used for the database. The second is a 3.6 TB storage for archive.

In order to support long pulse steady-state plasma, conventional CAMAC data interface was replace by the Yokogawa WE7000 and National Instrument COMPACT PCI systems. The original and obsolete O2 database was also replaced with ObjectStore object-oriented database [35, 68, 87]. Remote collaboration is achieved using standard network protocols on top of TCP/IP.

National Spherical Torus Experiment (NSTX)

National Spherical Torus Experiment (NSTX) [72] (1999 – present), which succeeds TFTR that ended in 1997, is a spherical magnetic fusion device constructed by the Princeton Plasma Physics Laboratory (PPPL) in collaboration with Oak Ridge National Laboratory, Columbia University, and the University of Washington. NSTX differentiates from traditional doughnut shaped tokamak fusion devices with its spherical tokamak design with a hole through the center of the device. NSTX researchers claim that the spherical design allows higher plasma confinement pressure, which produces more energy from the plasma. The design could also produce more stable plasma allowing miniaturization of fusion devices. An experiment pulse lasts about 0.5 second and requires about 10 minutes of cooling down between pulses [83].

The control system is composed of a combinations of control software and software toolkits [24, 83]. Most of the control system is integrated using the Experimental Physics and Industrial Control System (EPICS), which allows computers to control and read variables from physical devices through input/output controllers (IOCs) using EPIC's Channel Access (CA) protocol. EPICS is described in Section A.2.2. The most advanced control systems is the Plasma Control System (PCS) developed by General Atomics for the DIII-D experiment. PCS is a real-time digital system that controls the shape, position, and density of the plasma.

NSTX data management is based on MDSplus, which is a hierarchically organized database that is created for each NSTX shot. Both raw and analyzed data from the engineering and physics control systems are written into the 'shot tree'. Currently, MDSplus stores 100 MB of data per

shot, more than 5000 waveforms and 25,000 parameters. The NSTX MDSplus server runs on VMS. MDSplus clients run under VMS, UNIX, and Windows systems. There are FORTRAN, C, IDL, and Java interfaces to the MDSplus database.

In addition to data storage, MDSplus provides device control capability and an event system. For example, an MDSplus event can be used to refresh a plot with new data, or to control a CAMAC-interfaced instrument.

Wendelstein 7-X

Wendelstein 7-X [96] is a superconducting stellarator fusion device being constructed at the Max Planck Institute of Plasma Physics (IPP) in Garching, Germany, succeeding Wendelstein 7-AS, which operated from 1988–2002 [36]. Wendelstein 7-X, when completed, will be the largest stellarator fusion device in the world and expected to produce initial plasma in 2015. The goal of the device is to achieve long pulse, stead-state magnetic confined plasma for up to 30 minutes. The entire assembly consists of 50 non-planar, 20 planar superconducting coils, cryostat, plasma vessel, divertor and heating systems [7].

The computerized control system of Wendelstein 7-X utilizes a distributed and hierarchical design that consists of local control systems and central control systems. Local control systems control specific part or instrument, for example the plasma heating system, and is able to perform tests independently. The central control system coordinates local control systems during an experiment. The purpose of the central control system is to allow physics to specify experiment parameters through a central location [80].

Control processes with real-time requirements are controlled by fast control stations (FCS), which run VxWorks real-time operating system. Data acquisition and control (DAQ) subsystems

run Windows and Linux operating systems. Custom control software running on DAQ are written in Java programming language, which provides code portability (operating system independence). For data visualization, Siemens WinCC[2] was used. The control system interfaces with devices using Siemens programmable logic controllers (PLCs). Slow control processes use Siemens SIMATIC S7 PLCs.

A.2.2 Software Systems

MDSPlus

MDSPlus is a hierarchical data management software system developed by the Plasma Fusion Center at Massachusetts Institute of Technology in collaboration with the ZTH Group at Los Alamos National Lab and the RFX Group at CRN in Padua, Italy. It is specifically designed for managing control, research, and diagnostic data for fusion experiments. It is used by various plasma fusion research groups including Alcator C-Mod (MIT) and NSTX (Princeton).

Data in MDSPlus is stored hierarchically, where data for sub-components of an experiment are stored in different sub-trees. The reason for this design is to accommodate different experiment configurations and a wide variety of parts and instruments that produce data to be stored. Figure A.1 shows an example of a sub-tree in MDSPlus contains hierarchical data for a magnetics component. In addition to data management, MDSPlus also contain tools for data analysis and visualization.

Despite having a flexible data model, MDSPlus lacks powerful query capabilities. Most modern database systems support powerful query languages (such as SQL) and evaluation of queries. In MDSPlus, to specific data, one must write data traversal routines in IDL.

Experimental Physics and Industrial Control System (EPICS)

Experimental Physics and Industrial Control System (EPICS) [61] is an architecture and software framework for designing physics experiment control systems. According to the website,

[2]http://w3.siemens.com/mcms/human-machine-interface/en/visualization-software/scada/pages/default.aspx

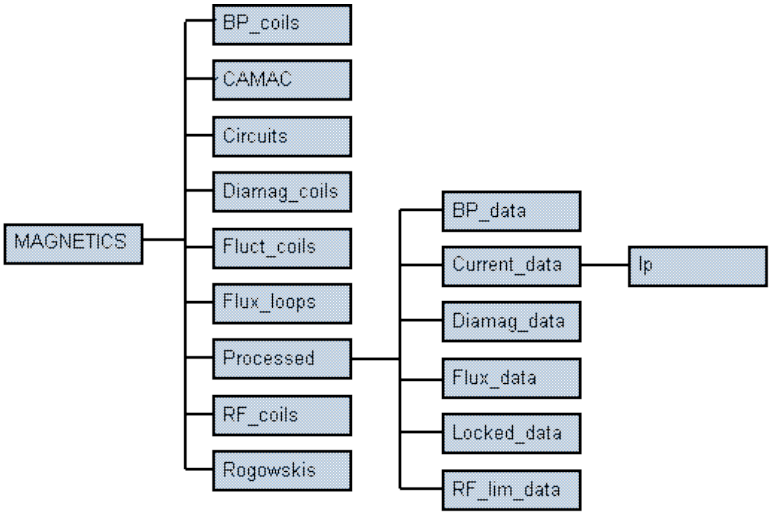


Figure A.1: MDSPlus magnetic sub-tree

"EPICS is a set of Open Source software tools, libraries and applications developed collaboratively and used worldwide to create distributed soft real-time control systems for scientific instruments such as a particle accelerators, telescopes and other large scientific experiments." Initially the system was called Group Test Accelerator Control System (GTACS), which was developed by Los Alamos National Laboratory (LANL), and was renamed to EPICS in 1991. EPICS is an open source software and being developed by Advanced Photo Source group[3] at Argonne National Laboratory[4] and Accelerator Technology group at Los Alamos National Laboratory (LANL)[5] [17, 47]. The following is a list of capabilities of the framework [47].

1. Remote control & monitoring of technical equipment
2. Data conversions and filtering
3. Closed loop control, both slow and fast
4. Access security
5. Equipment operation constraints

[3]http://www.aps.anl.gov/epics/index.php
[4]http://www.anl.gov/
[5]http://www.lanl.gov/

6. Alarm detection, reporting and logging
7. Data trending, archiving, retrieval and plotting
8. Automatic sequencing of operations
9. Mode and facility configuration control (save/restore)
10. Modeling and simulation
11. Data acquisition including image data
12. Data analysis

An EPICS control system is distributed in nature, which means that communications (read and write) with devices (e.g. power supply, and vacuum gauge) are achieved through standards computer network using UDP and TCP channel access (CA) protocol. Each device is controlled by a local input-output controller (IOC) which acts as a channel access server that publish process variables on a computer network. A process variable is a values of a device for example the current pressure of a vacuum chamber. Any part of the control system can read process variables through IOCs using channel access protocol which was built on top of UDP and TCP protocols. Control programs (also known as client programs) can also modify the states of the system buy modifying process variables. Figure A.2 shows an high-level view of the EPICS architecture, which consists of control software clients, IOC servers that control the devices and a computer network and channel access protocol that connect the components.

In addition, EPICS also consists of a collection of software for data analysis, data visualization, and monitoring. According to EPICS system developer Andrew Johnson, a typical control system can be constructed by piecing together existing software components [47]. Figure A.3 shows example control displays from the APD Linac, and Figure A.4 shows a screen capture of CSS-BOY from ITER.

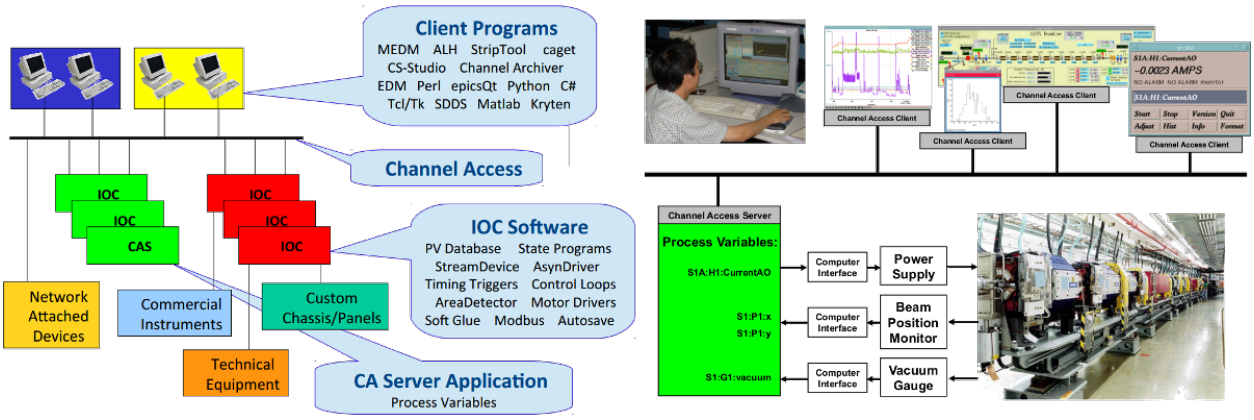


Figure A.2: EPICS architecture

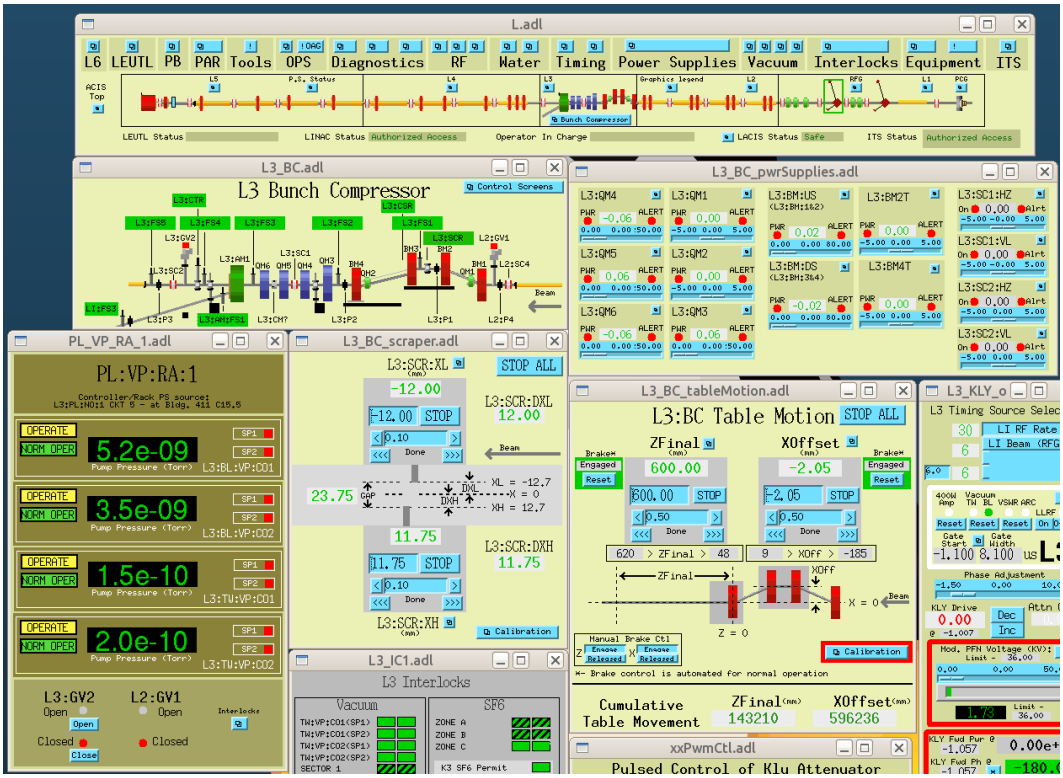


Figure A.3: MEDM displays for the APS Linac

A.3 MDPX Machine Configuration

The primary components of the MDPX device are: superconducting magnet, vacuum vessel and associated vacuum systems, plasma generation system, experimental control and data acquisition system, and plasma diagnostics. This section describes these components.

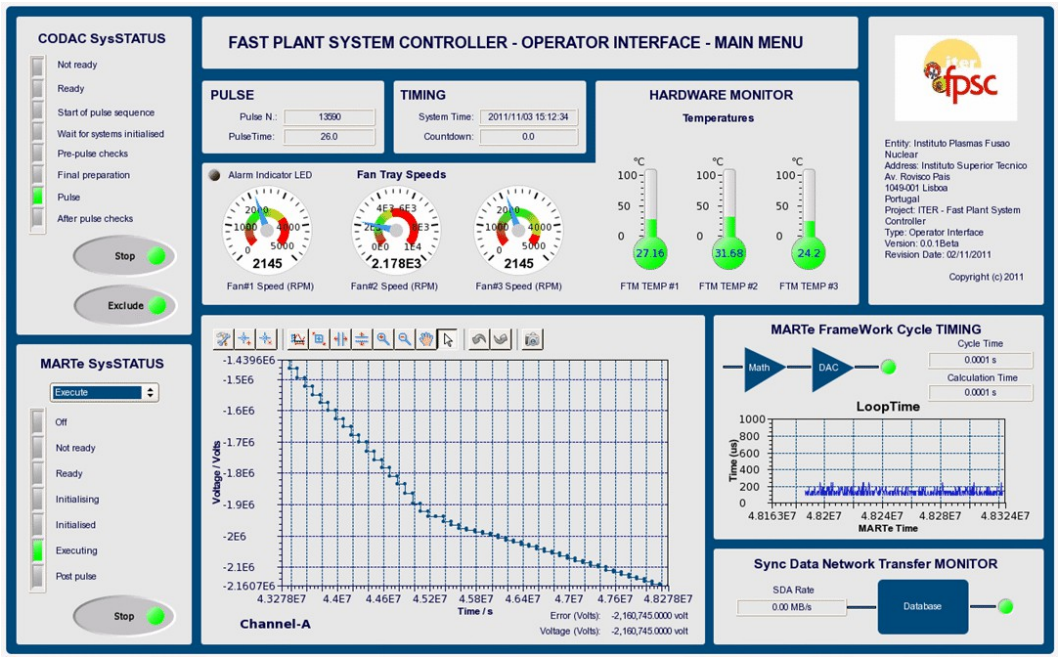


Figure A.4: CSS-BOY of ITER

A.3.1 Superconducting Magnet

The magnet assembly is an open-bore cryostat with a cylindrical warm bore that is 50 cm diameter and 157 cm axially. The assembly is divided into two parts separated by steel beams (Figure A.5(a)). Each of the two parts of the cryostat encloses a pair of two types of superconducting coils. Type 1 is the smaller coil that are located closer to the edge of the cryostat; while type 2 is the larger coil that are located closer to the center of the device. Each half contains one small and one large coil with the total of 4 coils in the entire assembly as shown in Figure A.5(b). The cryostat is evacuated and cooled to the temperature of 6.4 Kelvin, at which the coils becomes superconducting. The divided design of the magnet assembly provides a windows to allow diagnostic access to the plasma chamber. The coils and the cryostat were designed by Auburn University, the MIT Fusion Engineering group, and Superconducting Systems, Inc. (SSI) and manufactured by SSI in Billerica, Massachusetts.

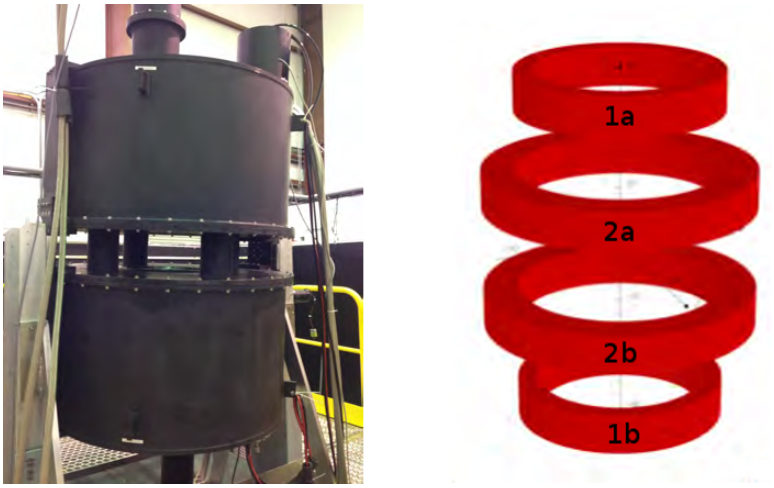


Figure A.5: (a) Complete magnet assembly which consists cryostat divide into two parts, each containing two superconducting coils. (b) Two types of superconducting coils and their shape [90].

To produce an uniformed magnetic field, a single TDK Lambda Genesys GEN8-180 power supply is used to provide current to all coils. To produce non-uniformed magnetic field configurations, three smaller power supplies are used to produce an 'anti-symmetric' configuration. In addition, several instruments are used to monitor the temperature of the magnet assembly. Lakeshore DT-670-SD silicon diode sensors are used to monitor radiation shields, while a more sensitive RX-202A-AA ruthenium oxide sensors are used to monitor the temperature of the coils. A Lakeshore Model 224 temperature monitor is interfaced with the data acquisition system through IEEE-488 interface (GPIB).

A.3.2 Vacuum Chamber and Vacuum Systems

The next major component is the vacuum chamber. The vacuum chamber is where plasma is produced and resides inside the open-bore of the magnet assembly. The device can be configured to house vacuum chambers of various shapes and dimensions. The chamber that is currently used has 8 sides (octagonal) with flat circular top and bottom as shown in Figure A.6 wit outer diameter of 43.2 cm (17 in) and a circular inner diameter of 35.5 cm (14 in), with opening of each face of dimension 12.7 cm (5 in) tall by 12.2 cm (5 in) wide.

Each face has the open dimension of 12.7 cm (5 in) tall by 12.2 cm (5 in) wide. The faces can be fitted with various flanges to adapt to diagnostic devices. Diagnostic devices monitor the parameters of the plasma and sends the data to the data acquisition system through Compact PCI Express interface. This is valuable data for researchers. A transparent window can also be fitted to allow video recording of the plasma inside the chamber.

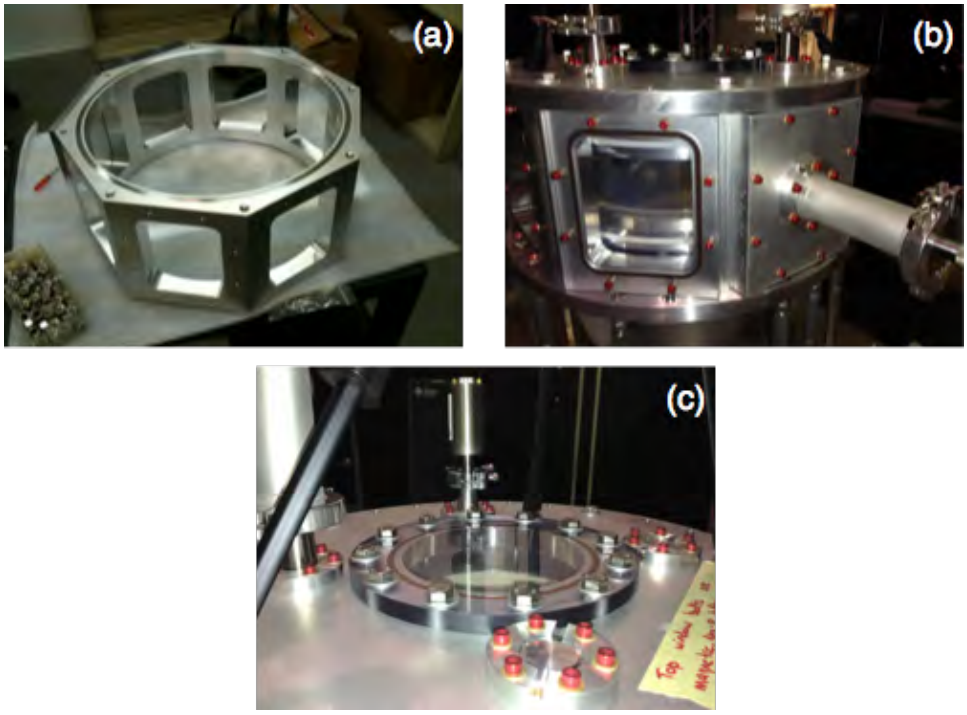


Figure A.6: Views of the MDPX vacuum chamber. (a) the octagonal frame of the vacuum chamber prior to adding the top and bottom flanges; (b) side view of the assembled vacuum chamber showing the top and bottom flanges, a window, and a side flange that is adapted to a QF40 nipple – the top and bottom electrodes used for plasma generation are also visible through the side window; (c) top view of the vacuum chamber showing a circular, top viewport and the adaptors to QF25 feedthroughs [90].

The devices that create a vacuum inside the chamber and deliver plasma gas into the chamber are controlled by the LavView control system, which is covered in Section A.4. A vacuum is created using an Agilent (Varian) Turbo V 81-T turbomolecular pump with an ISO63 inlet backed by an Agilent DS202 two-stage rotary vane pump. After the vacuum is established, a MKS 1179 mass flow controller with 100 standard cubic centimeter per minute (sccm) flow rate is used to

deliver argon gas into the chamber. At the present time, only a single type of gas is used to. A mixture of gases is planned in the future.

A.4 MDPX Control Software

The software portion of the system is roughly divided into two parts: the control software and data management software. The control software plays an major role in the entire experiment and is responsible for controlling the entire experiment process from controlling and monitoring experiment parameters to data acquisition and data storage. The control software interfaces and controls the data management portion of the system for data storage and archiving.

The data management software plays an subordinate role to the control software. The purpose of the data management software is to take the research data from the control software, store the data on disk, and catalog the data to make it search-able. An overview of the software components of the project is illustrated in Figure A.7.

This section describes these two parts of the control system in detail. The control software is described first, then the data management portion of the software system.

A.4.1 LabVIEW Control Software

The control and data situation software is a custom software running in LabVIEW [45]. Produced by National Instruments, LabVIEW is a programming and development platform specifically designed for engineering and scientific applications. Using this platform allows the research team to quickly design a control and monitoring software using existing packages and features provided by the platform. This control system runs on Windows Platform and is written to provide two functions: experiment control and monitoring, and data acquisition and storage. This section describe these functions and how they are realized.

Experiment control refers to controlling and setting the parameter of various devices and system that is required to produce plasma and run the experiment. These devices include the hardware components described in Section A.3, which include pneumatic valves, turbo-pumps for

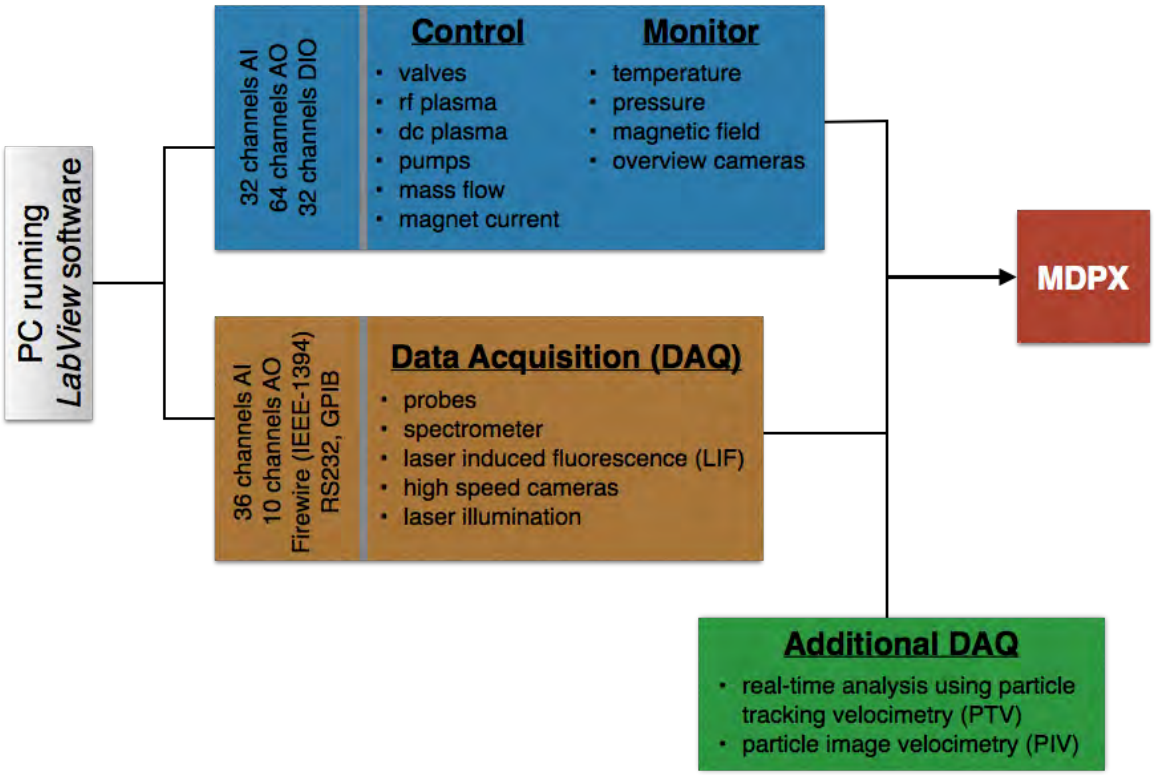


Figure A.7: Schematic layout of the experimental control/monitoring and data acquisition system.

creating the vacuum, current and voltage parameters for generating magnetic fields, and the RF and DC power supplies for plasma generation.

In addition to controlling the experiment, the control system also monitors the state of the experiment to ensure that the experiment is running within a tolerable range of parameters. The state includes the pressure inside the plasma chamber, magnetic fields, and temperature of the magnets. The control software communicates with various subsystem though National Instrument PXIe-1075 PXI Express chassis. Various communication cards are connected to the chassis that provides digital and analog input and output to the devices. The benefit of this control system is that it allows real-time feedback and summary of the experiment. Figure A.8 shows a snapshot of the front panel of the control software.

The second function of the control software is to coordinate data acquisition, which is the measurement and acquiring scientific data that is useful for research. Research data is acquired

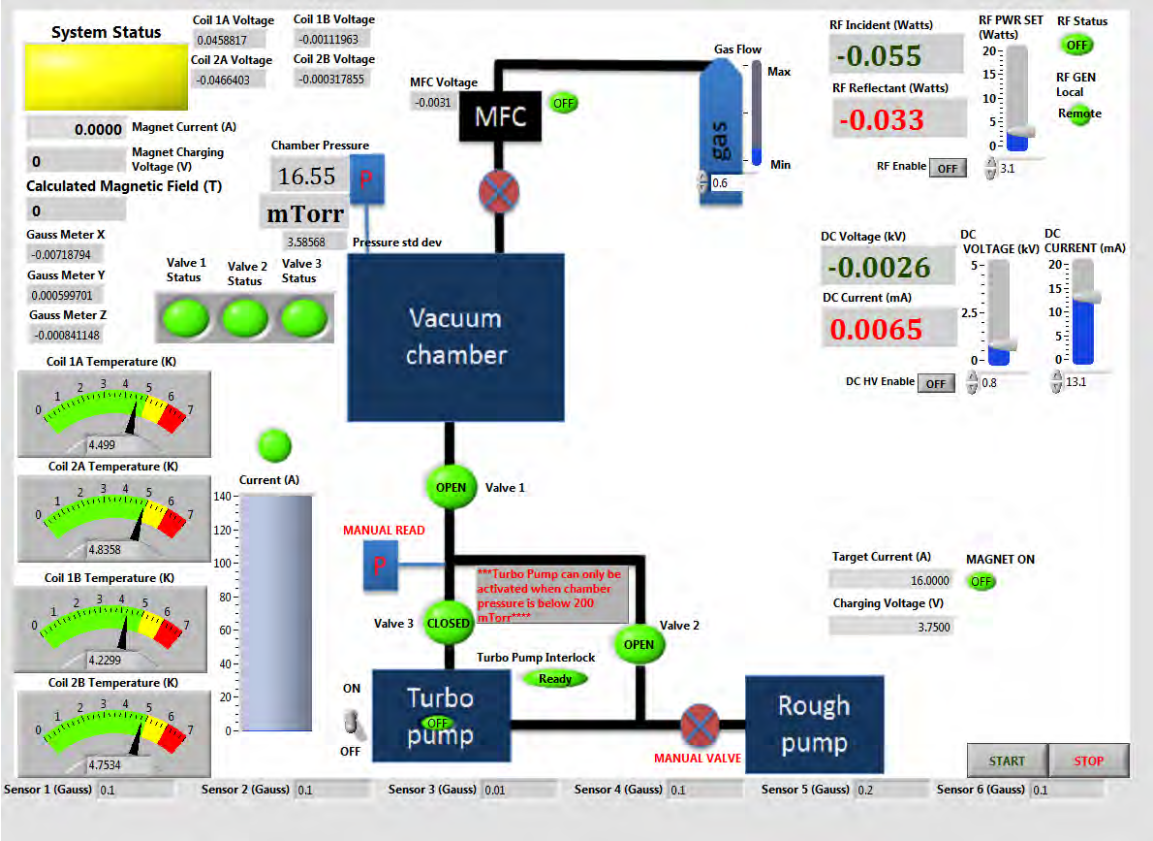


Figure A.8: A snapshot of the primary LabVIEW front panel from the control computer.

from diagnostic devices and subsystems. Diagnostics incorporated into the experiments are Langmuir probes, emission spectroscopy, laser induced fluorescence (LIF), and high speed cameras. In the same way as previously described, the control software talks with diagnostics though PXI communication cards connected to the PXIe-1075 PXI Express chassis. After the data had been acquired by the control system, it is stored on disk and handed to the data management portion of the system, which is described in the next subsection.

A.4.2 Database Management System

During an experiment, the control system gathers a bulk amount of data from diagnostics systems. These data are valuable to researcher and therefore need to be managed and stored so that the data can be easily retrieved by researchers, possibly at a much later time. In this section, we describe how data is stored and indexed in MDPX data storage system.

The bulk of the data gathered is stored on disk file system. The reason for this is that each instrument and diagnostics have different data format. Storing data on the file system allows us to easily store data files of different system and introduce new diagnostics. Like data stored in MDSPlus, the research data produced by the control software of MDPX are stored hierarchically. For example, for an experiment that has video and particle image velocimetry (PIV) diagnostics, the research data structure would contain a storage directory for videos and another for velocimetry, each containing its data files. [Research data structure].

In addition to the data gathered by the diagnostic systems, we also keep track of the machine configuration of each experiment. We use a MySQL[6] database for this purpose and contains the following configuration information.

1. Components: which components (parts) is used in an experiment. For example, the type of vacuum chamber, the electrodes, the type of diagnostics, such as probes and PIV.

2. Configuration: detailed information about how the components are interconnected.

3. Parameters: calibration and set-point information for each component used.

4. Summary: important metrics of the experiment.

5. Access Control: user and database access permissions.

An important role of the relational database is that it is the index of research data. A database record of an experiment contains the location where the bulk research data is stored on disk. This allows a researcher to quickly find all the data about a specific experiment. Due to its capability to quickly execute queries, the relational database allows researchers to compose queries based on key experimental metrics and find all experiments that matches the search criteria. For example, a researcher may search for experiments that used an octagon vacuum chamber with argon-40 gas, an Edmund A602FC 100 FPS camera, and a magnetic field of 1 Tesla.

[6]http://www.mysql.com/

The reason we choose a relational database for this purpose is threefold. The first is that relational database allows us to impose relationships among experiments and reuse existing information. For example, it is often the case that several experiments share the same machine configuration (components and parts used in a experiment), and only differ in the parameters of the plasma control devices, such power of electrodes and of the magnets. Our current database design allows us to share configuration records among experiments and store parameters separately. This reduces data volume and allows the database to perform queries without being slowed down by duplicate data.

The second reason is that MySQL is fast, stable, and supports broad range of data searching operations. SQL supports very powerful query capability and allows researchers to efficiently search experiment data. MySQL is an open source database management system that has been used by data data intensive products such as Facebook [91], YouTube [46], and Twitter [100]. In addition, according to a publication by Princeton Plasma Physics Laboratory on data management for Plasma Physics Databases [18], a relational database is several factors faster than MDSplus [85] database with reasonable data compression ratio.

Figure A.9 shows the database schema of the relation data, which is divided into 4 categories: catalog, setup, and summarized measurements. Tables in each category is described in Table A.1–A.3. In the remaining of this section, we briefly describe the purpose and structure of these tables.

Catalog

The *catalog* consists of a set of tables that keeps an inventory of parts and materials that we have at our facility and are used in the experiments. Table A.1 summarizes the purpose of the tables in this group.

Setup

The *setup* is the group of tables shown in the middle section of the schema shown in Figure A.2 (grouped in blue box). This group holds the bulk of the searchable metadata of the experiments

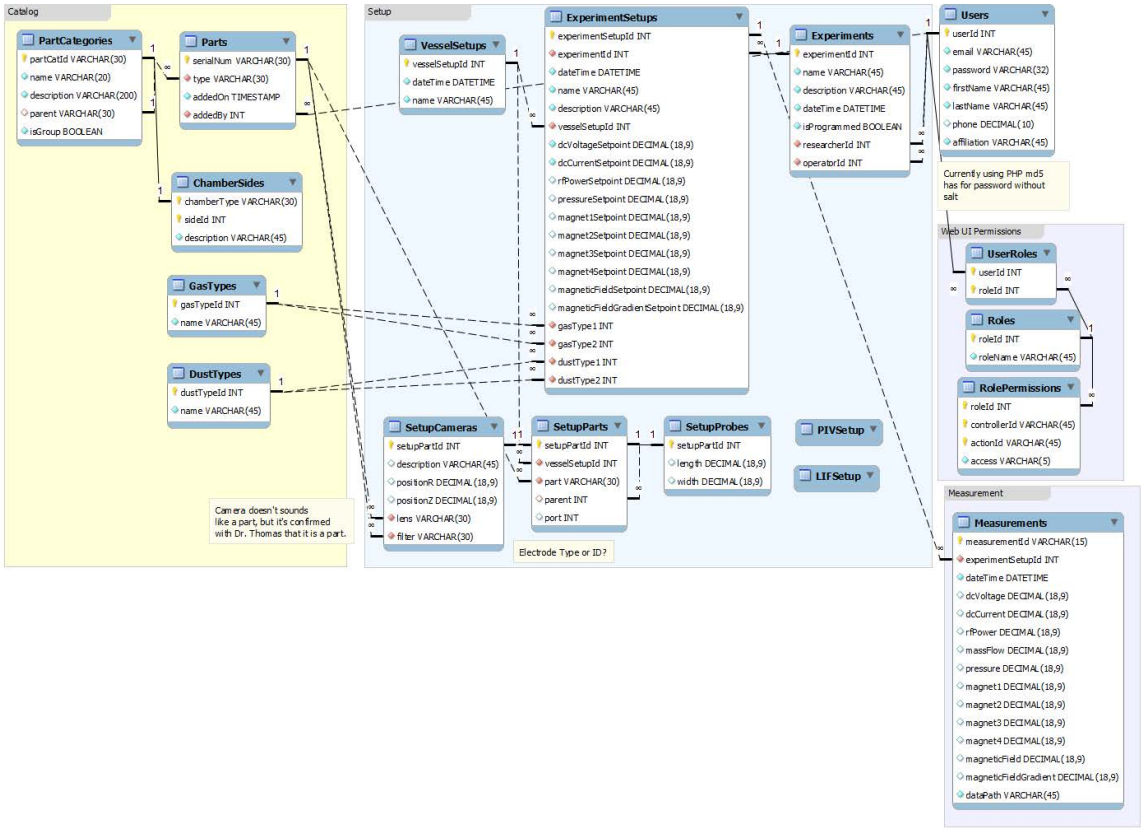


Figure A.9: Database schema.

Table A.1: Tables in Catalog

Table	**Description**
PartCategories	A hierarchy of parts. An example of a category is flanges, with side and top flanges as subcategories.
Parts	A catalog of experimental hardware parts. For example a flange, a hexagon chamber, or a camera.
ChamberSides	Enumerates the number of sides for part in Parts table that has the category of *chamber*.
GasTypes	A catalog of gases used for experiments, e.g. Argon.
DustTypes	A catalog of particles used for experiments, e.g. Silica 0.5.

and contains information of how an experiment is setup – which machine parts are used, how parts are connected, what are the set-points for instruments, and the grouping of experimental studies.

Table A.2 summarizes the purpose of the tables in this group.

Table A.2: Tables in Setup

Table	Description
VesselSetups	Each record of this table denotes a set of parts that makes a vessel, and the date the setup was constructed.
SetupParts	This table contains a list of parts for a vessel setup. This table references the parts in *Parts* table.
SetupCameras	This table contains a list of parts and parameters for a vessel setup that are cameras.
SetupProbes	This table contains a list of parts and parameters for a vessel setup that are probes.
PIVSetup	Reserved table currently not used.
LIFSetup	Reserved table currently not used.
ExperimentSetups	This table contains the set-points of instruments.
Experiments	This is a table for grouping a set of related runs. Each run would have set of experiment set-points. In essence, this table groups a set of entries in the ExperimentSetups table.

Summarized Measurements

The *Measurements* table is the only table in this category. For each run of experiment, there is a corresponding set of aggregated measurement data in the this table. The purpose of this table is to allow the researchers to perform queries to find the research data of interest.

Access Control and User Management

The last group are the user management and access tables. This category contains the *Users* table and the tables under the group named *Web UI Permissions*. These tables defines user roles and controls which role has access to and what action can be performed to different part of the database. Currently only the web interface is using this access control mechanism. Description of these tables follows (in Table A.3).

A.4.3 Web Interface

The web interface allows users of the system to view and, if permission granted, modify the information stored in the relational database without having to write complex data query sentences. A perfect example of the power of the web UI can be demonstrated via the *catalog*. Due to the

Table A.3: Tables in Setup

Table	Description
Users	Registered user that are allowed to login through web interface.
Roles	A list of user roles.
UserRoles	A list of role the users are assigned to. A user could be assigned to multiple rows.
RolePermissions	Action that can be performed on the web interface. For example, only users assigned with role of Admin can show the registered users ini the database.

hierarchical nature of the Part Categories (where one categories could have subcategories), it is cumbersome to insert or modify a category directly using SQL. For example, if a new part category is to be inserted using raw SQL, the user will first have to write a query sentence to find the identifier of the desired parent category; then, another query sentence to insert the new category, all the while there is room for human copying error.

Using the web interface, the user can easily see the hierarchical structure of the data as shown in Figure A.10, and to modify a category, just right click on a tree node, and a list of supported actions are shown.

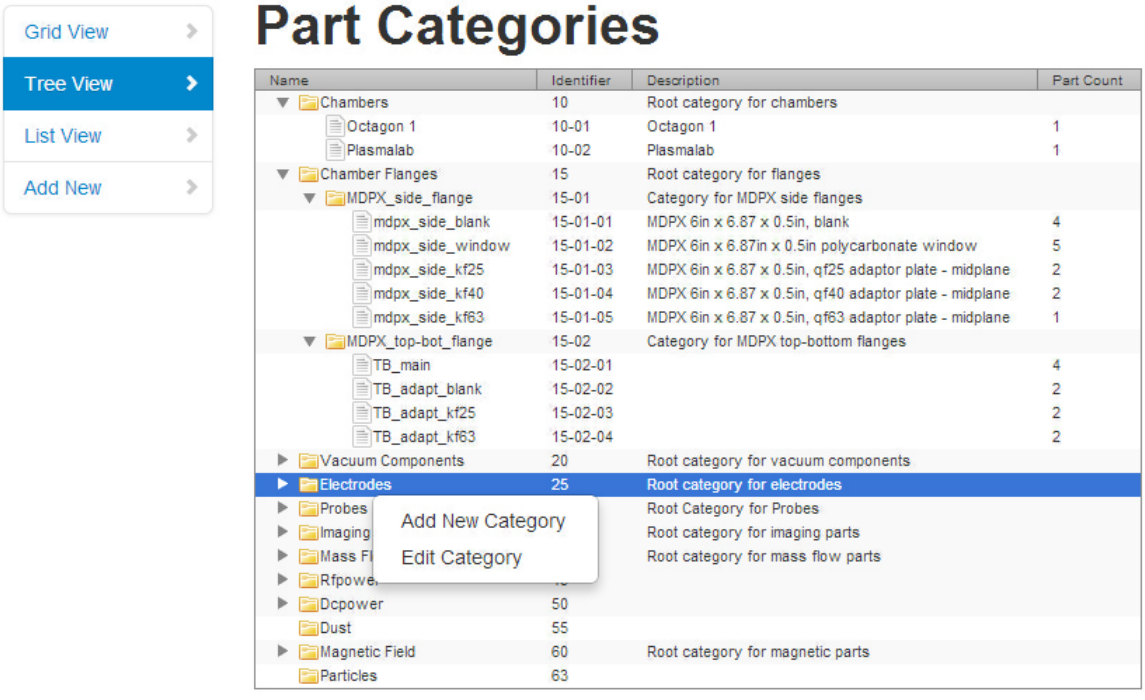


Figure A.10: Tree view of the part categories.

The web interface is implemented via PHP 5.3 with Yii Framework 1.1. The framework provides agnostic database access; therefore, RDBMS from different vendors can be used as the data store. The RDBMS that are supported include MySQL(what we use), PostgreSQL, SQLite, Microsoft SQLServer.

Inserting and Modifying Data

The interface facilitates entering new data into the database as shown in the Figure A.11.

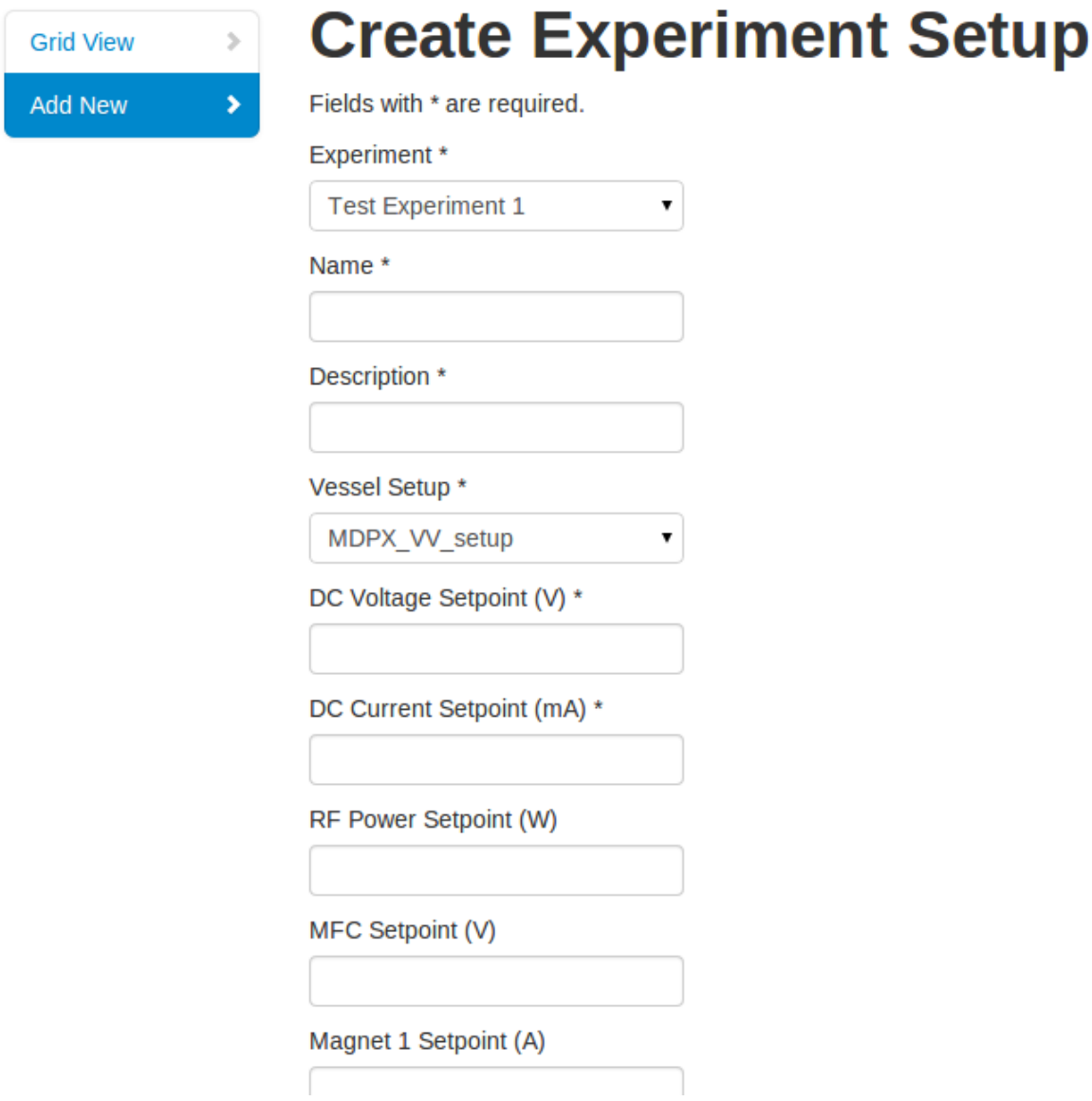


Figure A.11: Data entry form and data validation.

Two ways that the interface make entering and modifying data easier and less error-prone.

The first is that data input form provides selection drop-down menu for data item when possible. The content of these selection menus are usually populated from tables in the database. For example, to insert a new entry into the Experiments table (a experiment group), a researcher and operator user ID is needed. The data entry interface provides a selection menu for these fields so that the user does not have to perform additional step of looking up user IDs. When the data is submitted, the interface inserts the correct user IDs.

The second is that, the interface provides data validation. If a data item does not match a format or validation rule, the data is rejected and will not be saved to the database. The database also enforces data integrity rules. The interface adds a second layer of data integrity and constraint checks.

Data Filtering and Searching

Data filtering is supported in grid views. Figure A.12 shows the grid view of parts in the system. Users can specify filtering criteria on the columns at the top of the grid view. For string columns, the user can enter the complete or partial search string. For example, to search and researcher with the last name of "Thomas", the user can enter either "Thomas" or just "Thom" as filtering string. For numerical columns, comparison operators ($<$, $>$) can be used in filtering criteria.

A.5 Conclusion

In this chapter, we described a control and data acquisition system for the Magnetized Dusty Plasma Experiment(MDPX) system. As with any non-trivial system, the MDPX control is a tight integration between hardware and software. The control system is written in LabVIEW and is responsibly for controlling experiments and data acquisition. The control software controls hardware devices through NI PXI Express digital and analog input output cards connected to NI PXIe-1075

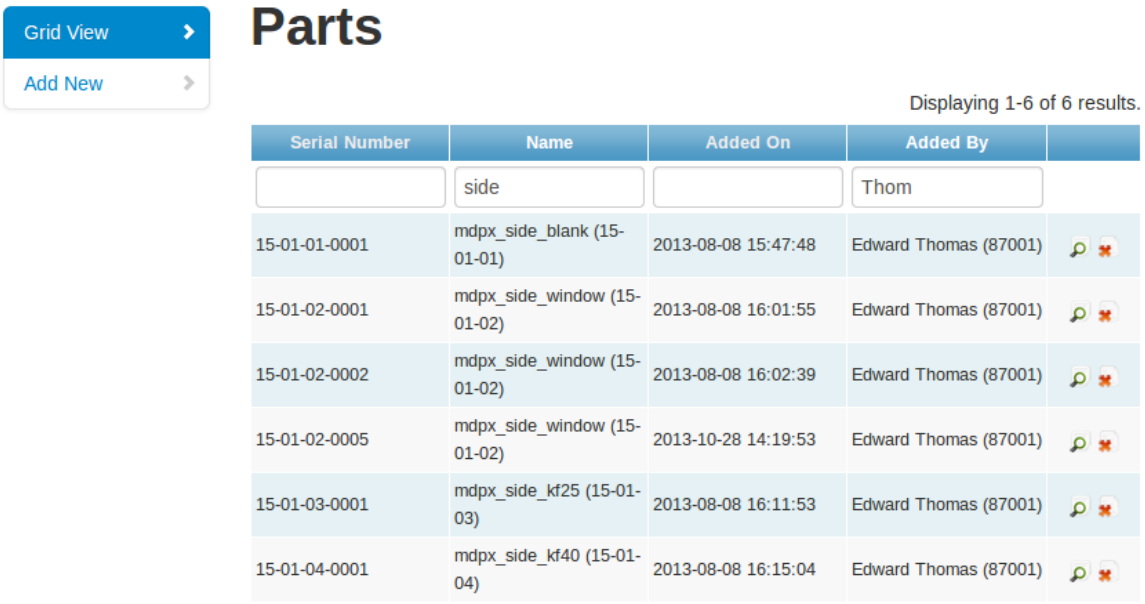


Figure A.12: Data Filtering.

PXI Express chassis. This configuration allows us to seamlessly integrate the software with hardware.

In addition to the control software, a relational database for cataloging experiments set up has been described. We choose to use MySQL for this purpose because it is the state-of-the-art database system and supports standard query language. We believe that our approach allows our team to quickly design and implement the system,allows the system to be easily maintained, and the system can be easily extended.

A.6 Database Table Reference

PartCategories Table

Column	Description
partCatId	Identifier of part category.
name	Display name of part category.
description	A textual description of this part category.
parent	Contains Part Category ID of the parent part category node.
isGroup	Denotes if a Part Category is an intermediate node or a leaf node.

Parts Table

Column	Description
serialNum	Unique identifier of each part. TWo physical parts of the same type in the catalog will have the same category ID, but different serial number.
type	Part Category ID.
addedOn	The date and time when the part is added to the catalog.
addedBy	The ID of the user who added the part to the catalog.

GasTypes Table

Column	Description
gasTypeId	Unique integer identifier.
name	Name of the gas, e.g. Argon.

DustTypes Table

Column	Description
gasTypeId	Unique integer identifier.
name	Name of the particle, e.g. Silica 0.5.

VesselSetups Table

Column	Description
vesselSetupId	Unique integer database record identifier.
dateTime	Date and time when this setup was created.
name	A descriptive name of this setup.

SetupParts Table

Column	Description
setupPartId	Unique integer database record identifier.
vesselSetupId	The ID of the vessel setup to which this part is attached.
part	Serial number of this part.
parent	The ID of the setup part to which this part is attached.
port	The location on the parent part on which this part is attached.

SetupCameras Table

Column	Unit	Description
setupPartId		The ID of the setup part of which this record is describing.
description		Description of the camera.
positionR	Degree	The R position of the camera.
positionZ	Centimeter	The Z position of the camera.

SetupProbes Table

Column	Unit	Description
setupPartId		The ID of the setup part of which this record is describing.
length	Centimeter	The length of the probe.
width	Centimeter	The width of the probe.

ExperimentSetups Table

Column	Unit	Description
experimentSetupId		Unique integer database record identifier.
experimentId		ID of an record in Experiments table.
dateTime		Date and time on which this setup was created.
name		Descriptive name.
description		Description.
vesselSetupId		Database ID of the vessel setup of this experiment.
dcVoltageSetpoint	Volt	
dcCurrentSetpoint	Milliamp	
rfPowerSetpoint	Watt	
pressureSetpoint	Volt	
magnet1Setpoint	Amp	
magnet2Setpoint	Amp	
magnet3Setpoint	Amp	
magnet4Setpoint	Amp	
magneticFieldSetpoint	Tesla	
magneticFieldGradientSetpoint	Tesla/Meter	
gasType1		
gasType2		
dustType1		
dustType2		

Experiments Table

Column	Description
experimentId	Unique integer database record identifier.
name	Descriptive name of the experiment group.
description	Description of the experiment group.
dateTime	The date and time on which this group was created.
isProgrammed	A boolean flag that signifies if this group of experiment is conducted by a program that iterates through a range of values for a set of paramenters.
description	Description of the experiment group.
researcherId	Foreign key to Users.userId. This means each record in this table has a researcher.
operatorId	Foreign key to Users.userId. This means each record in this table has an operator.

Measurements Table

Column	Unit	Description
measurementId		Primary key of this table.
experimentSetupId		Foreign key to ExperimentSetups.experimentSetupId. This relationship means every measurement is linked with one experiment setup, and every experiment setup has multiple measurements.
dateTime		Date and time when a record was recorded.
dcVoltage	Volt	Monitored average DC voltage of ...?
dcCurrent	Milliamp	Monitored average DC current of ...?
rfPower	Watt	
massFlow	Volt	
pressure	?	
magnet1	Ampere	
magnet2	Ampere	
magnet3	Ampere	
magnet4	Ampere	
magneticField	Tesla	
magneticFieldGradient	Tesla/Meter	
dataPath		Where the corresponding data is stored.

Users Table

Column	Description
userId	Primary key for this table.
email	Email address of this user.
password	Password of the user hashed using md5 without salt.
firstName	First name of the user.
lastName	Last name of the user.
phone	Contact telephone number of the user.
affiliation	Associated organization of the user.

Roles Table

Column	Description
roleId	Primary key for this table.
roleName	Name of this role, e.g. Admin.

RolePermissions Table

Column	Description
roleId	Foreign key to Roles.roleId. This means for reach
controllerId	Controller identifier of the web interface.
actionId	Action identifier of the web interface.
access	Have value of either ALLOW or DENY. This column specifies whether an operation identified by pair controllerId, actionId can be performed by a role.

Bibliography

[1] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*, pages 411–422, 2007.

[2] Wolf-Tilo Balke, Ulrich Güntzer, and Jason Xin Zheng. Efficient Distributed Skylining for Web Information Systems. In *EDBT*, pages 256–273, 2004.

[3] Paolo Bellavista, Axel Küpper, and Sumi Helal. Location-Based Services: Back to the Future. *IEEE Pervasive Computing*, 7(2):85–89, 2008.

[4] Jon Louis Bentley. K-d Trees for Semidynamic Point Sets. In *Symposium on Computational Geometry*, pages 187–197, 1990.

[5] K Blackler and AW Edwards. The JET Fast Central Acquisition and Trigger System. *Nuclear Science, IEEE Transactions on*, 41(1):111–116, 1994.

[6] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The Skyline Operator. In *ICDE*, pages 421–430, 2001.

[7] H-S Bosch, V Erckmann, Ralf WT Konig, Felix Schauer, Reinhold J Stadler, and A Werner. Construction of Wendelstein 7-X–Engineering a Steady-State Stellarator. *Plasma Science, IEEE Transactions on*, 38(3):265–273, 2010.

[8] Andreas Brodt, Daniela Nicklas, and Bernhard Mitschang. Deep integration of spatial query processing into native rdf triple stores. In *GIS*, pages 33–42, 2010.

[9] JR Burke and DJ Hollenbach. The Gas-Grain Interaction in the Interstellar Medium-Thermal Accommodation and Trapping. *The Astrophysical Journal*, 265:223–234, 1983.

[10] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *WWW*, pages 74–83, 2004.

[11] Baichen Chen, Weifa Liang, and Jeffrey Xu Yu. Progressive Skyline Query Evaluation and Maintenance in Wireless Sensor Networks. In *CIKM*, pages 1445–1448, 2009.

[12] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *VLDB*, pages 1216–1227, 2005.

[13] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, 1970.

[14] Douglas Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, jun 1979.

[15] Open Geospatial Consortium. OGC GeoSPARQL - A Geographic Query Language for RDF Data. http://www.opengeospatial.org/standards/requests/80, July 2011.

[16] Jia-Rui Cook. How Do We Know When Voyager Reaches Interstellar Space?, September 2013.

[17] Leo R Dalesio, MR Kraimer, and AJ Kozubal. EPICS Architecture. In *ICALEPCS*, volume 91, pages 92–15, 1991.

[18] W Davis and D Mastrovito. DbAccess: Interactive Statistics and Graphics for Plasma Physics Databases. *Fusion Engineering and Design*, 71(1-4):183–188, 2004.

[19] F Di Maio, L Abadie, C Kim, K Mahajan, P Makijarvi, D Stepanov, N Utzel, and A Wallander. The CODAC Software Distribution for the ITER Plant Systems. In *13th International Conference on Accelerator and Large Experimental Physics Control Systems*, volume 5, 2011.

[20] J Dietz and The ITER Joint Central Team. The ITER Fusion Experiment. *Vacuum*, 47(6):911–918, 1996.

[21] Raphael A. Finkel and Jon Louis Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.*, 4:1–9, 1974.

[22] Culham Centre for Fusion Energy. The Joint European Torus.

[23] Thomas W Fredian, JA Stillerman, and Martin Greenwald. Data Acquisition System for Alcator C-Mod. *Review of scientific instruments*, 68(1):935–938, 1997.

[24] D. Gates, M. Bell, J. Ferron, S. Kaye, J. Mendard, and D. Mueller. Initial Operation of NSTX with Plasma Control. In *EPS Conference on Control Fusion and Plasma Physics*, 2000.

[25] A Gibson. The JET Project. *Atom (London)*, 9(7), 1977.

[26] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Maximal Vector Computation in Large Data Sets. In *VLDB*, pages 229–240, 2005.

[27] CK Goertz and G Morfill. A Model for the Formation of Spokes in Saturn's Ring. *Icarus*, 53(2):219–229, 1983.

[28] CM Greenfield, GL Campbell, TN Carlstrom, JC DeBoo, C-L Hsieh, RT Snider, and PK Trost. Real-Time Digital Control, Data Acquisition, and Analysis System for the DIII-D Multipulse Thomson Scattering Diagnostic. *Review of Scientific Instruments*, 61(10):3286–3288, 1990.

[29] Donald A Gurnett, E Grün, D Gallagher, WS Kurth, and FL Scarf. Micron-Sized Particles Detected Near Saturn by the Voyager Plasma Wave Instrument. *Icarus*, 53(2):236–254, 1983.

[30] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD Conference*, pages 47–57, 1984.

[31] JongWoo Ha, Yoon Kwon, Jae-Ho Choi, and SangKeun Lee. Energy Efficient and Progressive Strategy for Processing Skyline Queries on Air. In *DEXA*, pages 486–500, 2009.

[32] Marios Hadjieleftheriou, Yannis Manolopoulos, Yannis Theodoridis, and Vassilis J. Tsotras. R-Trees - A Dynamic Index Structure for Spatial Searching. In *Encyclopedia of GIS*, pages 993–1002. Springer, 2008.

[33] Sohail Hameed and Nitin H. Vaidya. Log-Time Algorithms for Scheduling Single and Multiple Channel Data Broadcast. In *MOBICOM*, pages 90–99, 1997.

[34] Takahiro Hara. Cooperative Caching by Mobile Clients in Push-Based Information Systems. In *CIKM*, pages 186–193, 2002.

[35] Nakanishi Hideya, Kojima Mamoru, Ohsuna Masaki, Komada Seiji, Nonomura Miki, Yoshida Masanobu, Imazu Setsuo, and Sudo Shigeru. Steady-State Data Acquisition Method for LHD Diagnostics. *Fusion engineering and design*, 66:827–832, 2003.

[36] M Hirsch, J Baldzuhn, C Beidler, R Brakel, R Burhenn, A Dinklage, H Ehmler, M Endler, V Erckmann, Y Feng, et al. Major Results from the Stellarator Wendelstein 7-AS. *Plasma Physics and Controlled Fusion*, 50(5):053001, 2008.

[37] Gísli R. Hjaltason and Hanan Samet. Distance Browsing in Spatial Databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[38] Colin Hogben and FS Hogben. Interfacing to JET Plant Equipment Using the HTTP Protocol. *JDN/H (02)*, 11, 4 2008.

[39] S Horne, M Greenwald, I Hutchinson, S Wolfe, G Tinios, T Fredian, and J Stillerman. Performance of the C-Mod Shape Control System. In *Fusion Engineering, 1993., 15th IEEE/NPSS Symposium on*, volume 1, pages 242–245. IEEE, 1993.

[40] Chih-Hao Hsu, Guanling Lee, and Arbee L. P. Chen. A Near Optimal Algorithm for Generating Broadcast Programs on Multiple Channels. In *CIKM*, pages 303–309, 2001.

[41] IBM. IMS: Then and Today. http://idcp.marist.edu/pdfs/ztidbitz/22

[42] Atsuo Iiyoshi, Masami Fujiwara, Osamu Motojima, J Todoroki, N Ohyabu, and K Yamazaki. Design Study of the Large Helical Device. Technical report, National Inst. for Fusion Science, Nagoya (Japan), 1990.

[43] Atsuo Iiyoshi, A Komori, A Ejiri, M Emoto, H Funaba, M Goto, K Ida, H Idei, S Inagaki, S Kado, et al. Overview of the Large Helical Device Project. *Nuclear Fusion*, 39(9Y):1245, 1999.

[44] Tomasz Imielinski, S. Viswanathan, and B. R. Badrinath. Data on Air: Organization and Access. *IEEE Trans. Knowl. Data Eng.*, 9(3):353–372, 1997.

[45] National Instruments. LabVIEW System Design Software. http://www.ni.com/labview/, November 2015.

[46] Joab Jackson. YouTube Scales MySQL with Go Code. http://www.computerworld.com/article/2493815/database-administration/youtube-scales-mysql-with-go-code.html, December 2012.

[47] Andrew Johnson. Introduction to EPICS. http://www.aps.anl.gov/epics/docs/APS2014/01-Introduction-to-EPICS.pdf, September 2014.

[48] Iris A. Junglas and Richard T. Watson. Location-based services. *Commun. ACM*, 51(3):65–69, 2008.

[49] M. Kalish, P. LaMarche, M. Viola, J.L. Anderson, L. Ciebiera, R. Rossmassler, R.T. Walters, S. Langish, and M. Casey. Design Improvements and Lessons Learned for the TFTR Tritium Cleanup and Gas Holding Tank Sampling systems. In *Fusion Engineering, 1995. SOFE '95. Seeking a New Energy Era., 16th IEEE/NPSS Symposium*, volume 1, pages 556–559, Sep 1995.

[50] Michael Kamfonas. Recursive Hierarchies: The Relational Taboo! *The Relational Journal*, 13(6):377–387, 1992.

[51] Dave Kolas, Ian Emmons, and Mike Dean. Efficient Linked-List RDF Indexing in Parliament. In *International Workshop on Scalable Semantic Web Knowledge Base Systems*, pages 17–32, 2009.

[52] Dave Kolas and Troy Self. Spatially-Augmented Knowledgebase. In *ISWC/ASWC*, pages 792–801, 2007.

[53] Satoshi Konishi, Satoshi Nishio, and Kenji Tobita. DEMO Plant Design Beyond ITER. *Fusion engineering and design*, 63:11–17, 2002.

[54] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB*, pages 275–286, 2002.

[55] Manolis Koubarakis and Kostis Kyzirakos. Modeling and Querying Metadata in the Semantic Sensor Web: The Model stRDF and the Query Language stSPARQL. In *ESWC (1)*, pages 425–439, 2010.

[56] J.G Krom. The Evolution of Control and Data Acquisition at JET. *Fusion Engineering and Design*, 43(3-4):265–273, 1999.

[57] Wei-Shinn Ku, Roger Zimmermann, and Haixun Wang. Location-based Spatial Queries with Data Sharing in Wireless Broadcast Environments. In *ICDE*, pages 1355–1359, 2007.

[58] Wei-Shinn Ku, Roger Zimmermann, and Haixun Wang. Location-Based Spatial Query Processing in Wireless Broadcast Environments. *IEEE Trans. Mob. Comput.*, 7(6):778–791, 2008.

[59] H. T. Kung, Fabrizio Luccio, and Franco P. Preparata. On Finding the Maxima of a Set of Vectors. *J. ACM*, 22(4):469–476, 1975.

[60] Kostis Kyzirakos, Manos Karpathiotakis, and Manolis Koubarakis. Developing Registries for the Semantic Sensor Web using stRDF and stSPARQL. In *International Workshop on Semantic Sensor Networks*, 2010.

[61] Advance Photon Source Argonne National Laboratory. Experimental Physics and Industrial Control System. http://www.aps.anl.gov/epics, February 2015.

[62] Wang-Chien Lee and Dik Lun Lee. Signature Caching Techniques for Information Filtering in Mobile Environments. *Wireless Networks*, 5(1):57–67, 1999.

[63] Xuemin Lin, Yidong Yuan, Wei Wang 0011, and Hongjun Lu. Stabbing the Sky: Efficient Skyline Computation Over Sliding Windows. In *ICDE*, pages 502–513, 2005.

[64] Adam Machanic. *Expert SQL Server 2008 Development*. Apress, 2010.

[65] Dean Meltz, Rick Long, Mark Harrington, Robert Hain, and Geoff Nicholls. *An Introduction to IMS(TM): Your Complete Guide to IBM's Information Management System*. IBM Press, 2004.

[66] L Mestel and L Spitzer. Star Formation in Magnetic Dust Clouds. *Monthly Notices of the Royal Astronomical Society*, 116(5):503–514, 1956.

[67] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE Trans. Knowl. Data Eng.*, 13(1):124–141, 2001.

[68] Y Nagayama, M Emoto, H Nakanishi, S Sudo, S Imazu, S Inagaki, C Iwata, M Kojima, M Nonomura, M Ohsuna, et al. Control, Data Acquisition, Data Analysis and Remote Participation in LHD. *Fusion Engineering and Design*, 83(2):170–175, 2008.

[69] H Nakanishi, M Emoto, M Kojima, M Ohsuna, and S Komada. Object-Oriented Data Handling and OODB Operation of LHD Mass Data Acquisition System. *Fusion engineering and design*, 48(1):135–142, 2000.

[70] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[71] MB Niedner Jr and JC Brandt. Interplanetary Gas. XXII-Plasma Tail Disconnection Events in Comets-Evidence for Magnetic Field Line Reconnection at Interplanetary Sector Boundaries. *The Astrophysical Journal*, 223:655–670, 1978.

[72] Masayuki Ono, SM Kaye, Y-KM Peng, G Barnes, W Blanchard, MD Carter, J Chrzanowski, L Dudek, R Ewig, D Gates, et al. Exploration of Spherical Torus Physics in the NSTX Device. *Nuclear Fusion*, 40(3Y):557, 2000.

[73] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an Analysis of Range Query Performance in Spatial Data Structures. In *PODS*, pages 214–221, 1993.

[74] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive Skyline Computation in Database Systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[75] Dimitris Papadias and Yannis Theodoridis. Spatial Relations, Minimum Bounding Rectangles, and Spatial Data Structures. *International Journal of Geographical Information Science*, 11(2):111–138, 1997.

[76] Matthew Perry, Prateek Jain, and Amit P. Sheth. SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries. In *Geospatial Semantics and the Semantic Web*. Springer, 2011.

[77] P.-H. Rebut. ITER: The First Experimental Fusion Reactor. *Fusion Engineering and Design*, 30(12):85 – 118, 1995.

[78] P.H. Rebut, R.J. Bickerton, and B.E. Keen. The Joint European Torus: Installation, First Results and Prospects. *Nuclear Fusion*, 25(9):1011, 1985.

[79] N. R. Sauthoff, R. E. Daniels, and PPL Computer Division. TFTR Diagnostic Control and Data Acquisition System. *Review of Scientific Instruments*, 56(5):963–965, 1985.

[80] Jörg Schacht, Heike Laqua, Marc Lewerentz, Ina Müller, Steffen Pingel, Anett Spring, and Andreas Wölk. Overview and Status of the Control System of WENDELSTEIN 7-X. *Fusion Engineering and Design*, 82(5):988–994, 2007.

[81] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB*, pages 507–518, 1987.

[82] Padma K Shukla and AA Mamun. *Introduction to Dusty Plasma Physics*. CRC Press, 2010.

[83] Paul Sichta, J Dong, G Oliaro, and P Roney. Overview of the NSTX Control System. *arXiv preprint cs/0111055*, 2001.

[84] J.A. Stillerman, M. Ferrara, T.W. Fredian, and S.M. Wolfe. Digital Real-Time Plasma Control System for Alcator C-Mod. *Fusion Engineering and Design*, 81(1517):1905 – 1910, 2006.

[85] JA Stillerman, TW Fredian, KA Klare, and G Manduchi. MDSplus Data Acquisition System. *Review of Scientific Instruments*, 68(1):939–942, 1997.

[86] J. D. Strachan, H. Adler, P. Alling, C. Ancher, H. Anderson, J. L. Anderson, D. Ashcroft, Cris W. Barnes, G. Barnes, S. Batha, M. G. Bell, R. Bell, M. Bitter, W. Blanchard, N. L. Bretz, R. Budny, C. E. Bush, R. Camp, M. Caorlin, S. Cauffman, Z. Chang, C. Z. Cheng, J. Collins, G. Coward, D. S. Darrow, J. DeLooper, H. Duong, L. Dudek, R. Durst, P. C. Efthimion, D. Ernst, R. Fisher, R. J. Fonck, E. Fredrickson, N. Fromm, G. Y. Fu, H. P. Furth, C. Gentile, N. Gorelenkov, B. Grek, L. R. Grisham, G. Hammett, G. R. Hanson, R. J.

Hawryluk, W. Heidbrink, H. W. Herrmann, K. W. Hill, J. Hosea, H. Hsuan, A. Janos, D. L. Jassby, F. C. Jobes, D. W. Johnson, L. C. Johnson, J. Kamperschroer, H. Kugel, N. T. Lam, P. H. LaMarche, M. J. Loughlin, B. LeBlanc, M. Leonard, F. M. Levinton, J. Machuzak, D. K. Mansfield, A. Martin, E. Mazzucato, R. Majeski, E. Marmar, J. McChesney, B. McCormack, D. C. McCune, K. M. McGuire, G. McKee, D. M. Meade, S. S. Medley, D. R. Mikkelsen, D. Mueller, M. Murakami, A. Nagy, R. Nazikian, R. Newman, T. Nishitani, M. Norris, T. O'Connor, M. Oldaker, M. Osakabe, D. K. Owens, H. Park, W. Park, S. F. Paul, G. Pearson, E. Perry, M. Petrov, C. K. Phillips, S. Pitcher, A. T. Ramsey, D. A. Rasmussen, M. H. Redi, D. Roberts, J. Rogers, R. Rossmassler, A. L. Roquemore, E. Ruskov, S. A. Sabbagh, M. Sasao, G. Schilling, J. Schivell, G. L. Schmidt, S. D. Scott, R. Sissingh, C. H. Skinner, J. A. Snipes, J. Stevens, T. Stevenson, B. C. Stratton, E. Synakowski, W. Tang, G. Taylor, J. L. Terry, M. E. Thompson, M. Tuszewski, C. Vannoy, A. von Halle, S. von Goeler, D. Voorhees, R. T. Walters, R. Wieland, J. B. Wilgen, M. Williams, J. R. Wilson, K. L. Wong, G. A. Wurden, M. Yamada, K. M. Young, M. C. Zarnstorff, and S. J. Zweben. Fusion Power Production from TFTR Plasmas Fueled with Deuterium and Tritium. *Phys. Rev. Lett.*, 72:3526–3529, May 1994.

[87] S Sudo, Y Nagayama, M Emoto, H Nakanishi, H Chikaraishi, S Imazu, C Iwata, Y Kogi, M Kojima, S Komada, et al. Control, Data Acquisition and Remote Participation for Steady-State Operation in LHD. *Fusion engineering and design*, 81(15):1713–1721, 2006.

[88] Yufei Tao and Dimitris Papadias. Maintaining Sliding Window Skylines on Data Streams. *IEEE Trans. Knowl. Data Eng.*, 18(2):377–391, 2006.

[89] E. Thomas, R.L. Merlino, and M. Rosenberg. Design Criteria for the Magnetized Dusty Plasma eXperiment. *Plasma Science, IEEE Transactions on*, 41(4):811–815, 2013.

[90] E. Jr. Thomas, U. Konopka, D. Artis, B. Lynch, S. Leblanc, S. Adams, R. L. Merlino, and M. Rosenberg. The Magnetized Dusty Plasma Experiment (MDPX). *Journal of Plasma Physics*, FirstView:1–21, 3 2015.

[91] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1013–1020. ACM, 2010.

[92] Lewi Tonks and Irving Langmuir. Oscillations in Ionized Gases. *Physical Review*, 33(2):195, 1929.

[93] H Van der Beken, CH Best, K Fullard, RF Herzog, EM Jones, and CA Stead. CODAS: The JET control and data acquisition system. *IEEE Transactions on Nuclear Science*, 1987.

[94] World Wide Web Consortium (W3C). Resource Description Framework (RDF). http://www.w3.org/RDF/, February 2004.

[95] World Wide Web Consortium (W3C). SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/, January 2008.

[96] L Wegener. Status of Wendelstein 7-X Construction. *Fusion Engineering and Design*, 84(2):106–112, 2009.

[97] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB*, 1(1):1008–1019, 2008.

[98] Peter Weiss and Kate Ramsayer. Voyager 1 has Entered a New Region of Space, Sudden Changes in Cosmic Rays Indicate, March 2013.

[99] Wikipedia. Hierarchical Database Model. https://en.wikipedia.org/wiki/Hierarchical_database_model.

[100] Brett Winterford. Twitter, PayPal Reveal Database Performance. http://www.itnews.com.au/News/317811,twitter-paypal-reveal-database-performance.aspx, October 2012.

[101] William Van Woensel, Sven Casteleyn, Elien Paret, and Olga De Troyer. Mobile Querying of Online Semantic Web Data for Context-Aware Applications. *IEEE Internet Computing*, 15(6):32–39, 2011.

[102] J. Wong. Broadcast Delivery. *Proceedings of the IEEE*, 76(12):1566–1577, 1988.

[103] Wai Gen Yee and Shamkant B. Navathe. Efficient Data Access to Multi-Channel Broadcast Programs. In *CIKM*, pages 153–160, 2003.

[104] Baihua Zheng, Wang-Chien Lee, Ken C. K. Lee, Dik Lun Lee, and Min Shao. A Distributed Spatial Index for Error-Prone Wireless Data Broadcast. *VLDB J.*, 18(4):959–986, 2009.