Honours Year Project Report

Optimizing PostgreSQL for Social Network Databases

By

Huang Qi

Department of Computer Science and Department of Mathematics

School of Computing and Faculty of Science

National University of Singapore

2012

Honours Year Project Report

Optimizing PostgreSQL for Social Network Databases

By

Huang Qi

Department of Computer Science and Department of Mathematics

School of Computing and Faculty of Science

National University of Singapore

2012

Project No: H091570
Advisor: Prof Tay Yong Chiang
Deliverables:
    Report: 1 Volume
    Code: 1 CD

Abstract

Many social network services use an off-the-shelf database system like MySQL when they started up. By the time their service goes viral, they discover that the system is not scalable and has to be re-engineered.

There is now a relational schema, called RSN, that is tailored for social network applications. The purpose of this project is to examine how MySQL can be optimized when the schema has RSN's restricted form. For example, the search space for query optimization may be smaller, join size estimation may be sharper, semijoin reduction with MapReduce may be possible, new concurrency control may make ACID enforcement scalable, etc.

Subject Descriptors:
H.2.1 Database Management, Logical Design
G2.2 Graph Algorithms

Keywords:
Social Network, RSN, Database, PostgreSQL, Algorithm, Implementation

Implementation Software and Hardware:
PostgreSQL 9.1.3

Acknowledgement

I would like to thank Prof Tay, for his patient guidance and rigorous discussions. It gave me a lot of strength on my research. I would not be able to complete this project otherwise.

Also thanks for Zhou Jingbo and Bao Zhifeng, who spent a lot of time to discuss ideas on my research, explain to me on the background of RSN and give me a lot of encourage.

Last but not least, a lot of thanks to my friends for supporting me all the way.

Table of Contents

Chapter 1

Introduction

Tay, Bao and Zhou were doing research about designing a specific relational database schema on social network. They designed a schema, called RSN(Relational Schema for Social Networks), to generalize most of the features required for a social network product. The restriction of the database schema provides us an advantage to use a specific database system to improve the database performance. The relational database systems, open-source or commercial, have the design to support general queries, and general database schema. The programmers have to consider all situations in the process of a query, or in the backend processing of the database system.

For example, the query optimization has to consider general database schema, and the ACID property have to be enforced to ensure an overall satisfying performance. However, if we limit our thinking only within one schema, there can be a lot of savings. This project focuses on the query processing and find some re-design methods, and it will be implemented in PostgreSQL database system.

In the next several sections in this chapter, I will introduce the objectives of this research, the background of RSN and some basics of PostgreSQL. Chapter 2 is the introduction to join operator, including the definition and theorems of hash join and join tree. Chapter 3 will elaborate and establish some basic theorems on primary key join. Chapter 4 will discuss the theorems to find optimal join tree based on primary key join. Chapter 5 discusses some experiments on PostgreSQL to verify our theorems and the implementation of these theorems in PostgreSQL.

1.1 Objectives

The RSN schema is a good entry point for startups or small companies to start a social network. Tay et al. even worked out a tool for starting up online social network product (Tay, Bao, & Zhou, 2012). However, we can still further enhance this RSN system. We need a database system when we setup a social network. The most currently used system in many social network sites is MySQL, and there are also some other systems like Cassandra, Apache Hadoop that are used in different sites(Pingdom, 2010).

Relational database systems, like MySQL and PostgreSQL, have been used for decades and proved with stability and power. Facebook uses Cassandra and Hadoop at the same time to help scale and manage the huge dataset. MySQL and PostgreSQL can provide enough functionality for small company product. But as the company grows, there will be more scalability issues emerging.

On the other hand, building a database system of MySQL with Cassandra and Hadoop, or to do modifications to MySQL to fit for the fast scale-up, is expensive for small company, as the technological barrier is high. They also need to hire more technicians to build up the system and the maintenance will also require more people. With RSN, we can attempt to change the implementation of MySQL or PostgreSQL to be more suitable for social network product to provide with more scalability and stability.

The query processing, especially query optimization, in relational database system is complicated and consumes a lot of time and space. The optimal query plan in many database system, like PostgreSQL(PostgreSQL, 2012d), Oracle(Antoshenkov & Ziauddin, 1996) and Microsoft SQL Server(Shankar, Nehme, Aguilar-Saborit, Chung, Elhemali, Halverson, Robinson, Subramanian, DeWitt, & Galindo-Legaria, 2012), is found using System R method(Chaudhuri, 1998). It is a variant of dynamic programming, which only considers left-deep plan and avoids Cartesian product. The left-deep plan first joins two base tables together, then join the intermediate result with another base table, and so on. A Cartesian product is a construction to build a new set out of a number of given sets. Each member of the Cartesian product corresponds to the selection of one element each in every one of those sets. As System R algorithm considers only left-deep plan and avoids any Cartesian product, the generated plan may not be optimal and sometimes is quite bad in performance.

The query optimizers in existing DBMSs often suffer from intolerably long optimization time or poor optimization results when optimizing large join queries(Tao, Zhu, Zuzarte, & Lau, 2003). Given the study of paper by Tao et al. in the snowflake(Tao et al., 2003), we are quite inspired by their research of rewriting snow-schema queries by heuristic rules. In a special form schema, there may be some special properties we can make use.

Besides, the paper from Scheufele and Moerkotte(Scheufele & Moerkotte, 1997), shows two efficient algorithms of generating left-deep trees possibly containing cross products for chain queries. It coincides with the idea of Tao's paper and shows a good example of developing and using heuristics.

Studying the properties of RSN, we can find the useful ones and seek for some heuristics to help design efficient algorithms for query processing. For example, we can find some properties of RSN to find the optimal join trees. If we can find any property of RSN schema to establish fast optimization, there will be a great saving on some complex queries for the social network products.

This project initially aims at modifying the query optimizer for MySQL. That is mainly because most of the existing social networks use MySQL, as it is light, fast and stable. However, PostgreSQL is used more on research and study area. The community of PostgreSQL is also very active and there are a lot of contributions from the mailing list. We therefore choose PostgreSQL for better implementation for this undergraduate research. If in the future, the idea on RSN is working well and to be realized as product, we can still migrate the modification to MySQL.

The first part of this research is to find potential ways of modifying PostgreSQL to improve its efficiency. This requires a mathematical analysis of optimal strategies for joining relational tables. It will also take the most time of the research. The second part is to implement the modifications inside PostgreSQL and evaluate their performance.

1.2 About RSN

When setting up the database for a social network product, people usually begin with a data model. Intuitively, most people would regard the social network as a graph. The nodes represent the people, and the edges can describe the relationship between them, as shown in Figure 1.1.(Infosthetics, 2008)

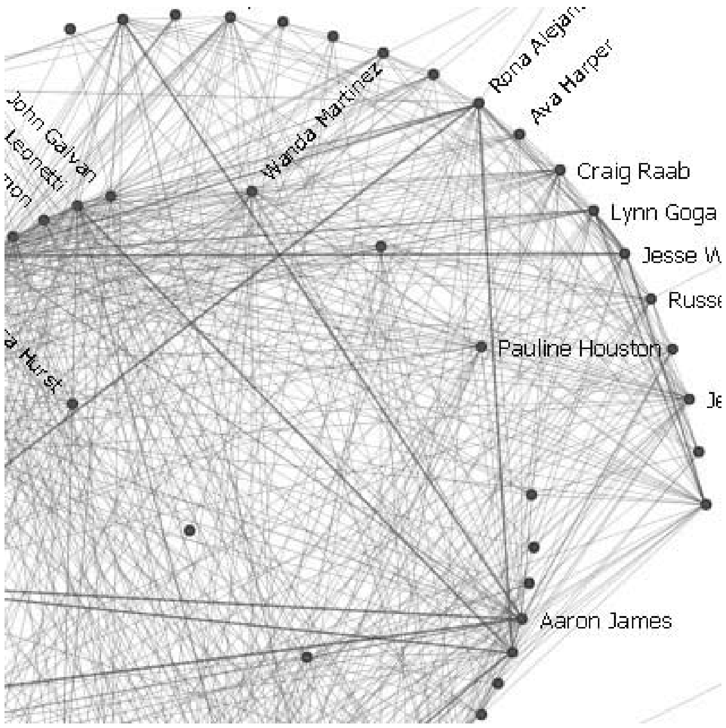


Figure 1.1: A social network graph

However, there are many disadvantages for this representation: it cannot describe the dynamics of the social network; it is very hard to include the attributes of members, like gender, address, photos and so on; a graph model is navigational and that makes it hard to run queries that cannot be formulated as graph traversal (Tay, Bao, & Zhou,). Graphs are useful when we come to investigate the relationships in social networks, but very tedious to express other properties of social networks. The current social network research will obtain the dataset from social network published dataset, which is extracted from MySQL or PostgreSQL, then transferred to a graph for corresponding analysis, as shown in 1.2.

However, restricting the schema of the relational dataset of social network might also achieve the same effect as that in a graph, as in Figure 1.3. We would like to skip the process of extracting

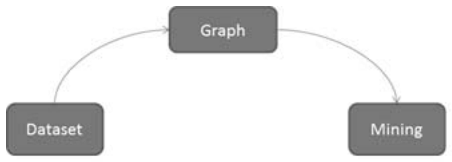


Figure 1.2: Mining 1

data to a graph. If we can do the analysis on the dataset directly, it will be more intuitive and grant full access to the raw data. It also provides us designs of new, multi-dimensional algorithms that have insights not available to graph algorithms.

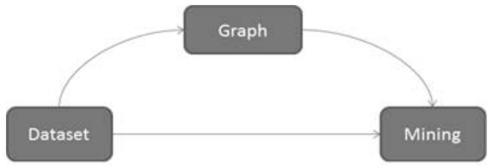


Figure 1.3: Mining 2

The schema designed for RSN is presented as the Figure 1.4.

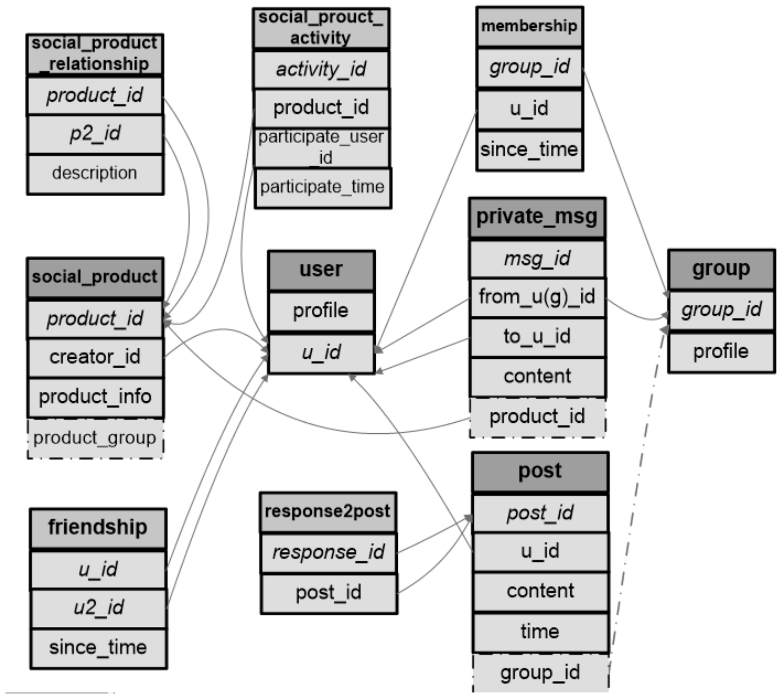


Figure 1.4: RSN: a relational database schema for social networks

It has a table for each of 6 basic entities:

- user is generic; it can be Jim, an advertiser, a retailer, etc.

- original_post is anything that is produced by the user, such as a blog, a photograph, an advertisement, a video, etc.
- response2post may be a comment on a blog, a response to a comment, writing on a FaceBook wall, etc.
- group records details (names, membership size, etc.) of an interest group.
- private_msg is a message that is visible only to sender and receiver(s).
- social_product may be a coupon generated by a commercial user and consumed by customers, an event created for a group, a poll among some users, etc.

In addition, there are tables for 3 relationships:

- friendship records a Facebook friendship, a Twitter follow relation, a DBLP co-authorship, etc.
- membership connects a user and a group.
- social_product_activity connects a user and a social product through an activity (buy a coupon, vote in a poll, etc).

The RSN schema is designed to fit in most of the services that are provided in social network. A social network product can build up their database according to this schema and store all the information. There are also some sample analyses according to some specific social networks, like ACMDL, coupon dissemination. From the experiments with respect to several graph analysis operations, for example the community structure, cluster detection, action propagation, we can see there are many analyses of graph can be done in RSN schema.

Let us look at an example of instantiation of RSN schema. In Figure 1.5, we model the social network implicitly in the ACM digital library(ACMDL) dataset through RSN.

user is instantiated as author. **post** is instantiated as papers; we use the optional group_id to identify the publication venue. **social_product_relationshipis** instantiated as citation, because citation indeed represents a particular relationship between the social product paper.

friendship is instantiated as coauthorship; the attribute paper_id is a foreign key that references papers, identifying the paper that was co-authored. **group** is instantiated as conference/journal. The table papers is a generalization that covers both journal and proceeding papers. (Tay et al.,)

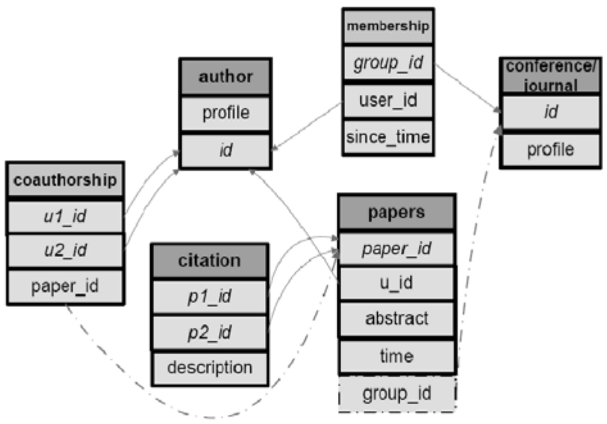


Figure 1.5: An example of RSN instantiation for the ACM Digital Library

1.3 Introduction to PostgreSQL

PostgreSQL is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. It is fully ACID compliant(atomicity, consistency, isolation, durability), with full support for foreign keys, joins, views, triggers, and stored procedures (in multiple languages).(PostgreSQL, 2012c)

Next we will introduce some knowledge about the work flow of PostgreSQL. After the PostgreSQL server is started, the client connects to backend server called postmaster via TCP or UNIX domain socket and communicates via frontend-backend protocol. Upon query is submitted in the client, the backend starts the query processing. There are three steps for a query to be transformed to the standard query tree — Lex and parse, analysis, and rewriter. After these three steps, the query tree produced will be sent to query optimizer and a query plan will be generated and then evaluated to return the result set.

A good plan is chosen by its cost, which is estimated by evaluating the I/O with the database statistics gathered by ANALYZE. The keyword geqo_threshold is used as a number to determine whether to use genetic query optimization to do planning, defaulted as 12. If the number of tables to be joined is smaller than this number, the query optimizer will use System R algorithm, otherwise use genetic algorithm. System R algorithm is a deterministic, near exhaustive planner algorithm. Genetic algorithm is a heuristic optimization method which operates through randomized search which can achieve a better performance (PostgreSQL, 2012b). To simplify the measurement for now, we consider only System R algorithm. System R is a variant of dynamic programming search algorithm. It makes use of the heuristics that

- no Cartesian product can exist in an optimal plan
- only left-deep plan will be considered
- early selection and projection should apply.

The steps of System R algorithm are just the basic dynamic programming approach with the use of the above assumptions. The complexity for the system R algorithm is pseudo-polynomial.

The objective of our research is to find a way to better optimize a query under RSN schema. If we find a feasible algorithm, we will make modification on the optimizer. We might also need to do modification on the language definition(handled by parser, analyzer, and rewriter) to create new query language to query on a specific query plan, which means we can specify a query plan in the new language and PostgreSQL backend will execute that plan.

Chapter 2

Join Operator

In a relational database, there are relations, also called tables. Each relation has its relational schema that is a set of attributes. An attribute has an attribute type, like integer, text, date and so on. A relation join combines two relations based on their common attributes. Assume we have two relations R and S, the **join** of R and S, is written as $R \bowtie_\theta S$, where θ is a join predicate.

The join predicate links the common attributes from two relations by equality or inequality, thus it has two types, equality predicates or inequality predicates. The inequality predicates only applicable to comparable attribute types, like integer. Join can be viewed as a cross-product of two relations followed by selections with the join predicate, as follows:

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

where σ denotes **selection**, which is to select some rows from the relation inside the bracket satisfying predicate θ.

A join operation with equality join predicate is called equijoin. If a join is an equijoin and the join attribute of one relation is its primary key and the join attribute of the other relation is the foreign key referencing the first relation's primary key, then the join is a join on primary key.

In this and later chapters we study the properties of primary key join and how it can help on join query optimization and execution.

2.1 Hash Join

The performance for join is a large concern for a database system, as usually a join takes a big portion of performance cost.

There are several algorithms for executing a join query, nested loop, block nested loop, sort-merge join, hash join and for each join method above, they have an index-based join. For nested loop and block nested loop, when they process large size relations, it will be quite costly as it will iterate the outer relation a big number of times, which requires many disk I/Os. In this research, we consider only hash join.

Hash join identifies partitions in relation R and S in a partitioning phase and, in a subsequent probing phase. The idea is to hash both relations on the join attribute, using the same hash function h. If we hash each relation (ideally uniformly) into k partitions, we are assured that R tuples in partition i can join only with S tuples in the same partition i. Then we can read in a (complete) partition of the smaller relation R and scan just the corresponding partition of S for matches. We never need to consider these R and S tuples again.

However, there is a potential risk. If the R partition is not evenly distributed on join attribute, and the hash function h does not partition R uniformly, the hash table for one or more R partitions may not fit into memory during the probing phase. It is called *partition overflow*. We handle this problem by recursively applying the hash join technique to the join of the overflowing R partition together with the corresponding S partition.

We now analyze the cost estimation for hash join. For example, we are given two relations R and S. **Here and in the later sections, we just denote the size of a relation by its small symbol, for example, relation $\boldsymbol{R}$ has size r (just small letter for the relation symbol)**. In the first phase of hash join, we need to read in R and S, and then write out their partitions, the cost is thus 2(r+s) I/Os. In the probing phase, if the partition overflow does not happen, the cost is (r+s) I/Os, ignoring the cost of writing out final result set (we can not improve on this). So the total cost is 3(r+s) I/Os.

To be simple, we restrict our study to hashjoin in the later chapters. In the future, we still need to study all cases of join method, but that will take more complicated analysis.

2.2 Join Tree

We first need to define a full binary tree. A **tree** can be defined as recursively a collection of nodes. The nodes contained in another node is called the **subnode** or **child** of its **parent node**. The children and their children and so on can be called the **descendants** of a node. A tree can be drawn as a graph by representing every node as a point and connect it with its subnode by a line. For example, $\{\{A, B, C\}, D, \{E, F\}\}$ is a tree, and the drawing is shown in figure 2.1. A node with no subnode is called a **leaf**, otherwise it is called a **internal node**. The line connecting the node and its subnode is called an **edge**. The node that is not a subnode of any node is called the **root**. If we want to look for a particular node, we usually start from the root, and traverse the edges downward until we find that node. The number of edges we traverse from the root to the particular node, is called the **level** of that node. A tree with all nodes having at most two subnodes is called a **binary tree**. If for any internal node, it has exactly two subnodes, then this tree is called a **full binary tree**. In a full binary tree, a node is just a pair of subnodes, if we treat it as a tuple, then the order counts, we call the subnode on the left a **left subnode** or **left child**, the subnode on the right a **right subnode** or a **right child**. We call a node together with all its descendants and the edges, a **subtree**. If the subtree's root is a left(right) child of a node, it can also be called the **left subtree**(**right subtree**) of the node.

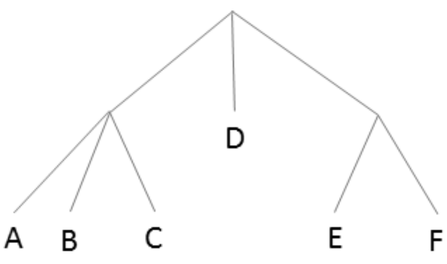


Figure 2.1: An example of tree, representing $\{\{A, B, C\}, D, \{E, F\}\}$

A join query may have several *join plans*. The relation join order and the join methods for each join determines the join plan. For example, $R \bowtie S \bowtie T$ have join plan of ((R hashjoin S) hashjoin T), (R mergejoin (S hashjoin T)) and so on. Executing different join plans will incur different costs. Most relational database systems will have a query optimizer to choose the optimal join plan. In order to choose the optimal join plan, the system needs to estimate the cost of each plan.

One common method for estimation is by **selectivity**, or **reduction factor**. The selectivity for an operator is defined as the proportion of tuples of the operand relations that participate in the result of the operation. Thus, the selectivity for a join of relation R and S, denoted sel_{RS} is

$$sel_{RS} = \frac{|R \bowtie S|}{r \times s} \tag{2.2.1}$$

where $|R \bowtie S|$ is the size of result set of joining R and S.

We need to estimate selectivity in order to estimate the result set size, which is reverse of the above equation. The estimation equation depends on the join predicate as follows

$$sel(A = value) = \frac{1}{|\pi_A(R)|}$$
$$sel(A > value) = \frac{max(A) - value}{max(A) - min(A)}$$
$$sel(A < value) = \frac{value - min(A)}{max(A) - min(A)}$$

where, $\pi_A(R)$ is projection of R to columns A and keeps only distinct values. $max(A)$ denotes the maximum value of column A in R. $min(A)$ is the minimum value of column A in R.

Usually, we can visualize the join plan by full binary tree, called **join tree**. As shown in Figure 2.2. This graph shows a join plan of seven relations R, S, T, M, N, P and Q and the join tree corresponds to join plan of $(((R \bowtie S) \bowtie T) \bowtie M) \bowtie (N \bowtie (P \bowtie Q))$.

A join tree decides the join order of a simple or complex join query. We can evaluate a join tree top-down by recursively evaluating a node. Starting from the root, if the left or right subtree is an internal node, we evaluate that subtree, or it is a leaf, then we join the left and right subtree results. This is using recursion on join tree evaluation. You may look at Figure 2.2 as an example. In order to evaluate this tree, we need to evaluate left subtree and right subtree, which both are non-leaf. Then we evaluate them as tree again, untill we come to leaves, like $R \bowtie S$.

However, we can also view the tree evaluation bottom up. We look at the tree, and find the simple subtrees, which means a subtree with only two leaves, like $R \bowtie S$ and $P \bowtie Q$, then

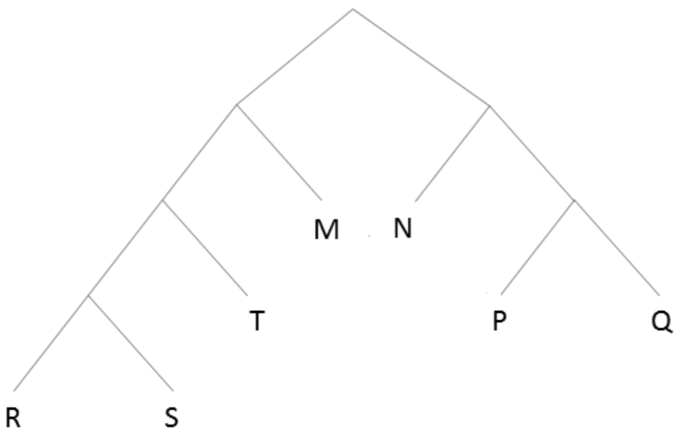


Figure 2.2: A join tree example which shows join plan of $(((R \bowtie S) \bowtie T) \bowtie M) \bowtie (N \bowtie (P \bowtie Q))$

we evaluate from those two subtrees and go up. In the above join tree example, we compute $R \bowtie S$ first, then the intermediate relation joins with T, then with M, lastly with a imtermediate relation by joining N with $P \bowtie Q$.

The **intermediate relation** is the result of joining some subtrees. For example, $((R \bowtie S) \bowtie T)$ will produce a relation with all attributes of R, S and T. And an intermediate relation will have the associated **intermediate table size**.

Chapter 3

Joins on Primary Key

RSN schema, as shown in Chapter 1.2, has primary key joins on all the relations. Suppose, we have relations R and S, if an attribute in S is a foreign key referencing the primary key in R, then we denote that as $S \rightarrow R$.

For a query that joins several relations, they will preserve a topology for their referencing relationship. For example, if a query joining relations R, S, T and M, these four relations may have referencing structure as $R \rightarrow S \rightarrow T \leftarrow M$. The above structure is one topology, while if we check another four relations in that schema, say M, N, P and Q, and their structure might be $M \rightarrow N \rightarrow P \rightarrow Q$, this is another topology, with all arrows pointing in one direction.

As we can see, for three relations, there will be 3 topologies. Assume we have three relations R, S and T, we have $R \rightarrow S \rightarrow T$, $R \rightarrow S \leftarrow T$ and $R \leftarrow S \rightarrow T$. The first topology has all arrows pointing to one direction. The second one has arrows pointing inward. The last one has arrows pointing outward. Other structures can map to one of the three topologies.

For four relations, there will be non-linear topologies. Suppose four relations R, S, T and M, there may have a non-linear topology $\begin{tikzcd} R \arrow[r] & S \arrow[r] \arrow[d] & T \\ & M & \end{tikzcd}$.

We may also have cyclic topology. For example, $\begin{tikzcd} R \arrow[r, shift left] \arrow[r, shift right] & S \end{tikzcd}$ is a cyclic topology. This can happen, if S is a relation with composite primary key, and R is referencing the primary key.

We have some definitions for primary key joins.

Definition 3.0.1. If we have two relations with $R_1 \rightarrow R_2$, then we call R_1 the **referencing**

relation, R_2 the **referenced relation**. We say R_1 is **referencing** R_2 and R_2 is **referenced by** R_1. And the relation that is only referencing some relations or only referenced by some relations is called a **terminal relation**.

Definition 3.0.2. We say two relations are **linked** if they have a referencing relationship. Any relation that is linked with two or more relations a **link relation**, and the relation that is linked with only one relation an **end relation**.

Definition 3.0.3. If we have n relations, the **distance** between two relations is the number of relations between them.

For example, we have 5 relations with topology $R_1 \rightarrow R_2 \leftarrow R_3 \rightarrow R_4 \rightarrow R_5$. R_1 is an end referencing terminal relation. R_2 is a link referenced terminal relation. R_3 is a link referencing terminal relation. R_4 is a referenced relation, and also a referencing relation. R_5 is an end referenced terminal relation. The distance between relation R_1 and R_4 is 2, that of R_2 and R_3 is 0.

Definition 3.0.3 holds as we consider only non-cyclic join topology for now. If there is a cycle in the join topology, the number of relations between two relations in the cycle is not well-defined. Because of the limit of time, this report only studies non-cyclic topologies. **For the later chapters, we will not consider the cyclic topology case**.

Definition 3.0.4. We say two relations are **connected** if they have a referencing path connecting them. A topology is **connected** if any two relations in the topology are connected.

Theorem 3.0.5. *A join tree with unconnected topology contains Cartesian product.*

Proof. Induction Basis: For two relations that are not connected, we can only join them in Cartesian product.

Induction Hypothesis: Assume the statement is true for $n, n-1, ..., 2$ relations.

Induction Step: We need to prove the statement is also true for $n+1$ relations. Now we have $n+1$ relations and the topology is not connected, and can be divided into two sets, which are not connected. The join tree must have two subtrees, A and B. If the left subtree relations are all from set A, and the right subtree relations are all from set B, then the two subtrees are not connected, and the root join is Cartesian product. If the left subtree has some relations from set A, and some from set B, then by induction hypothesis, the left subtree has Cartesian product. $\square$

3.1 Formation of Primary Join Tree

We generally try to avoid Cartesian product in the join tree. Cartesian product, in most cases, will cause very large cost on the join evaluation.

For primary key join, we could avoid Cartesian products quite easily. For a join tree, if the left subtree and right subtree have no link, which means there is no relation in the one subtree referencing the primary key of any relation in the other subtree, then the join is a Cartesian product. The join tree in Figure 3.1 is an example, the join between subtree of $R \bowtie S$ and M has no link, thus is a Cartesian product.

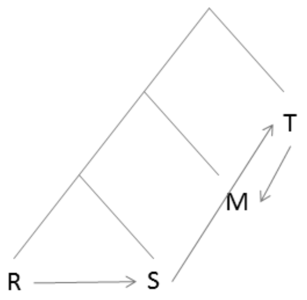


Figure 3.1: A join tree containing Cartesian product $((R \bowtie S) \times M)$

Lemma 3.1.1. *For a primary key join topology, assume there is no Cartesian product in the join tree. Then there must be a referencing relationship between the two subtrees.*

Proof. In primary key join, the relations are either joined on primary key or on Cartesian product. If there is no referencing relationship between the two subtrees, there is no possible primary key join between the two subtrees, there is only Cartesian product. So there must be a referencing relationship between the two subtrees. □

We only consider the join tree having a link(a referencing relationship) between its left subtree and its right subtree. As for our hashjoin consideration, if we switch the two subtrees, the join cost should be the same. Without loss of generality, for join with relations $\{R_1, R_2, R_3, ..., R_n\}$ in a linear topology, we just put the part with smaller index in the left subtree.

Let's have a look at Figure 3.1 again. The topology in the figure is $R \to S \to T \to M$, but the left subtree joins $\{R, S, M\}$ and the right subtree is T. Our next theorem says that any join tree for topology $R \to S \to T \to M$ must have subtrees of $\{R\}$ and $\{S, T, M\}$, or $\{R, S\}$ and

$\{T, M\}$, or $\{R, S, T\}$ and $\{M\}$.

Theorem 3.1.2. *For a join query with relations $\{R_1, R_2, ..., R_n\}$, a join tree without Cartesian product divides the relations into two sets, one set forms the left subtree, and the other set forms the right subtree. Each set of relations is still having a connected topology. The two subtrees are joined by the referencing relationship connecting these two topologies.*

Remark 3.1.3. If the relations are in a linear topology, then we can divide the relations into two sets, one set has relations $\{R_1, R_2, ..., R_i\}$, for some integer $i < n$, and the other has relations $\{R_{i+1}, R_{i+2}, ..., R_n\}$. Because only so, the two sets can be connected. And we will put subtrees formed by relations $\{R_1, R_2, ..., R_i\}$ in the left subtree, and the other as the right subtree, as shown in Figure 3.2.

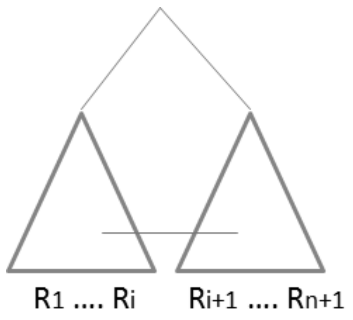


Figure 3.2: The formation of join trees

Proof. As any subtree with unconnected topology will contain Cartesian product, by Theorem 3.0.5. Thus the sets of relations in both subtrees have to be in connected topology.

The two subtrees have to be connected by some referencing relationship between the two subtrees, by Lemma 3.1.1. We claim there is only one such referencing relationship. If there are two, and one referencing relationship connecting R_i from the left subtree and R_k from the right subtree, the other one connecting R_j from the left subtree and R_p from the right subtree. Inside the left subtree relations, the relations are connected, then R_i and R_j are connected, and the same for right subtree, so R_k and R_p are connected. So there is a cycle. We are only considering non-cyclic case, so between the two subtrees there is only one referencing relationship, and the two subtrees are linked by this relationship. □

For example, we pick the link of $R_2 \leftarrow R_3$ in $R_1 \rightarrow R_2 \leftarrow R_3 \rightarrow ... \rightarrow R_{n-1}R_n$, and divide the group into $\{R_1, R_2\}$ and $\{R_3, R_4, ..., R_{n-1}, R_n\}$. Then those two subgroups can form the two subtrees accordingly.

We can try to find how many join trees we can have for a n-relation join query. Say, we have $f(n)$ join trees for a n-relation join query, considering only linear topology. Thus, we have $f(n) = f(1)f(n-1) + f(2)f(n-2) + f(3)f(n-3) + ... + f(n-2)f(2) + f(n-1)f(1)$ for $n \geq 3$, with f(1)=1, f(2)=1. Even we easily eliminate the Cartesian product join trees, we can see there are still at least $f(n) \sim \Omega(2^n)$ join trees.

Corollary 3.1.4. *Assume the number of non-Cartesian join trees for a n-relation join query in linear topology is $f(n)$. Then $f(n) \sim \Omega(2^n)$.*

Proof. $f(n) \sim \Omega(2^n)$ means there exists some c such that $f(n) \geq c \times 2^n$ for any integer $n \geq 3$. We take $c = \frac{1}{4}$ and prove this by induction.

Induction Basis: $f(3) = 2 \geq \frac{1}{4} \times 2^3$.

Induction Hypothesis: This statement is true for $n \geq 3$, that is $f(n) \geq \frac{1}{4} \times 2^n$.

Induction Step: $f(n+1) > f(1)f(n) + f(n)f(1) = 2f(n) \geq 2 \times \frac{1}{4} \times 2^n = \frac{1}{4} \times 2^{n+1}$. □

For the later sections, we will assume the condition implies no Cartesian product on any join tree, if not specifying otherwise. Theorem 3.1.2 will be used implicitly in many places.

3.2 Primary Join Selectivity

From now on, we will discuss joins as joining on primary key, and use the join topology to express the status of how the relations are referencing one another.

Join on primary key has a very important, but easy to see property.

Theorem 3.2.1. *Suppose we have two relations* R *and* S*, and they have referencing topology* $\mathrm{R} \rightarrow \mathrm{S}$*, then the size of result set of joining* R *and* S *is just r.*

Proof. This theorem is very easy to verify. As the join attribute of R is referencing the primary key of S, which means in S, this attribute has no duplicate, and also every attribute value of the join attriubte in R has a value in S. Thus after joining, every tuple of R will join with exactly one tuple in S, so the tuples in R are neither replicated nor deleted. So the size of the join result set is the same as the size of r. □

Next, let us have a look at the selectivity in the primary key joins.

Corollary 3.2.2. *For a simple two relation join with* R *and* S, *if* R → S, *then* $sel(\mathrm{R} \bowtie \mathrm{S}) = \frac{r}{r \times s} = \frac{1}{s}$.

Proof. This comes from the definition of selectivity, Equation 2.2.1. □

The Theorem 3.2.1 will be a very strong and useful theorem if we can extend it to more general cases. For example, we have a set of relations of topology $R \to S \to T \to M \to N$, then the cost of any join tree of a query joining these relations can be easily estimated. As shown in Figure 3.3, the intermediate table size is very easy and accurate to calculate.

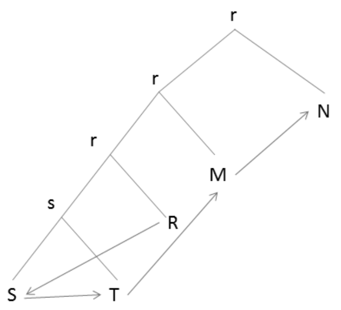


Figure 3.3: Estimation for intermediate tuple sizes, using Theorem 3.2.1

It is strongly desirable for our estimation to be accurate. On top of the accuracy of estimation of selectivity, we can have more accurate estimation of the cost of each join tree and thus find a better join tree. Besides, in the traditional estimation of selectivity, which is based on the assumption that the values for the join attribute are uniformly distributed, the estimation may have a large error if the distribution is non-uniform. Furthermore, in practice, the choice of join tree has a big impact. Different join trees may have very large difference in the running time.

However, Theorem 3.2.1 does not suffice for join size estimation if the topology has a relation that is referenced by more than one relation. For example, if we have three relations which have topology $R \to S \leftarrow T$, we consider all the join trees. There are two join trees, listed in Figures 3.4 and 3.5.

If we just use our estimation, in the first case, the final result size is r, while in the second, the final result size is t! The choice of join tree should not make any difference to the final result size. Otherwise, the join trees are ill-defined. Changing the join tree computes a different result, but the underlying query can have one and only one result.

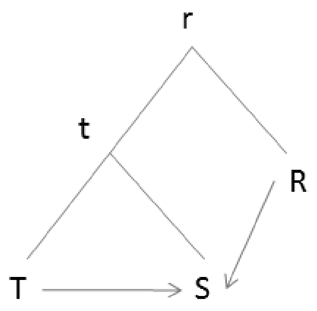


Figure 3.4: Join tree 1 for topology $R \rightarrow S \leftarrow T$

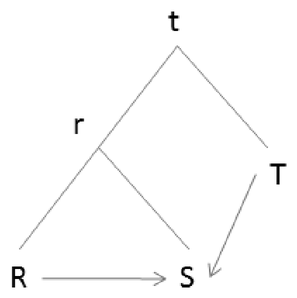


Figure 3.5: Join tree 2 for topology $R \rightarrow S \leftarrow T$

Let us look at Figure 3.4 again. For the intermediate relation $T \bowtie S$, name it P, the selectivity of this join is $\frac{1}{s}$, and the intermediate tuple size is $t \times s \times \frac{1}{s} = t$. But for $P \bowtie R$, what is the selectivity? As $T \rightarrow S$, after the join, the primary key inside S is no more a primary key, unless the join attribute in T is also a primary key. Thus in the second join, the primary key join no longer holds! We cannot directly use Theorem 3.2.1 to calculate.

Since Theorem 3.2.1 no longer applies, we have to come back to selectivity. Let us assume the primary key values are distributed uniformly in the foreign keys. Then we have the following theorem.

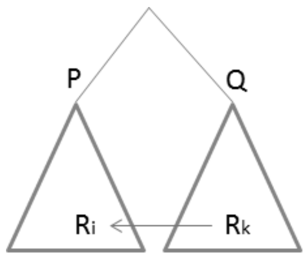


Figure 3.6: Primary key join

Theorem 3.2.3. *If we have n relations, then for a join tree shown in Figure 3.6, if the value of foreign key of R_k in Q is distributed uniformly, the selectivity of the final join operation is* $\frac{1}{r_i}$.

Proof. For the right subtree, in the final relation Q, the foreign key of R_k referencing primary key of R_i is distributed uniformly. Then for every value of foreign key, there are $\frac{q}{r_i}$ tuples having this value. After joining P and Q, there will then be $p \times \frac{q}{r_i} = \frac{pq}{r_i}$ tuples. According to Equation 2.2.1, the selectivity is $\frac{\frac{pq}{r_i}}{pq} = \frac{1}{r_i}$. □

As we know R is referencing S inside $T \bowtie S$(name it P) in Figure 3.4, according to Theorem 3.2.3, the selectivity of $R \bowtie P$ should be $\frac{1}{s}$. We want to use this selectivity to do estimation of intermediate table size. The final result set size for Figure 3.4 is $t \times r \times \frac{1}{s} = \frac{tr}{s}$. And for Figure 3.5, it is $r \times t \times \frac{1}{s} = \frac{tr}{s}$, which is the same as that of Figure 3.4. Using this selectvitiy equation, for any join tree, the estimations of their final result size will be the same. As Figures 3.7 and 3.8 shows, we can see for the two join trees, the estimations for their intermediate table size and final result set size agree with each other. For the left subtrees of both graphs, as they have the same join relation set $\{R, S, T, M\}$, their final intermediate table size is the same, both $\frac{rm}{t}$.

However, the assumption in Theorem 3.2.3 is very restrictive. We are assuming the value of foreign key of R_k in Q is distributed uniformly. In order to calculate the intermediate table sizes as in Figure 3.7 and Figure 3.8, we need the assumption to be established on every subtree. It is quite hard to do the check on every subtree, and we are checking on the joined result set of every subtree, which should be avoided in estimation of cost. If we could just assume the uniform distribution is holding for every foreign key, it would be much easier. That is, if every foreign key has its values uniformly distributed, then for every subtree P, and some relation inside P is referencing another some relation in another subtree, the referencing relation's foreign key value is uniformly distributed in the final join set of P. But this statement may not be true. For example, we have in Q three relations with topology $A \rightarrow B \leftarrow C$, and their tables are shown in Figure 3.9. C1 in table A is referencing another relation. We can see, Col1 and Col2 in A are uniformly distributed on their values, and Col2 in C is also uniformly distributed on its values. But in the joining result, shown in Figure 3.10, we can see, Col1 is not uniformly distributed on its values any longer.

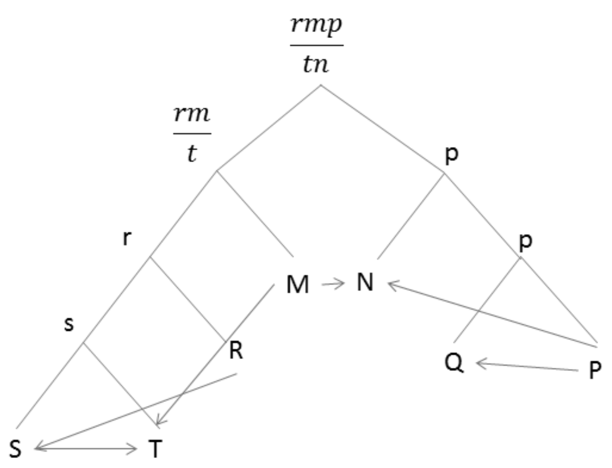


Figure 3.7: Selectivity calculation using Theorem 3.2.3 for one join tree of $R \rightarrow S \rightarrow T \leftarrow M \rightarrow N \leftarrow P \rightarrow Q$

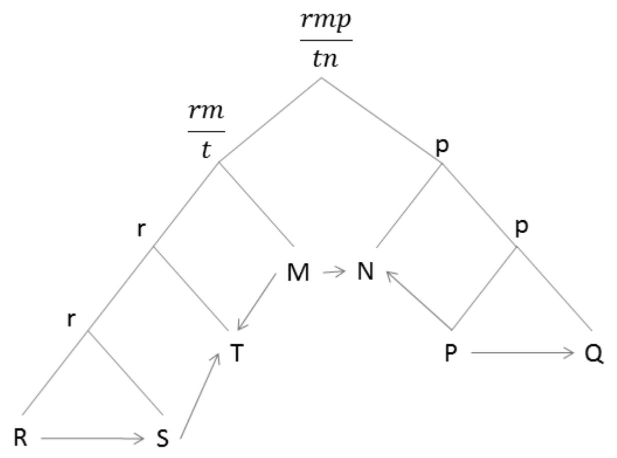


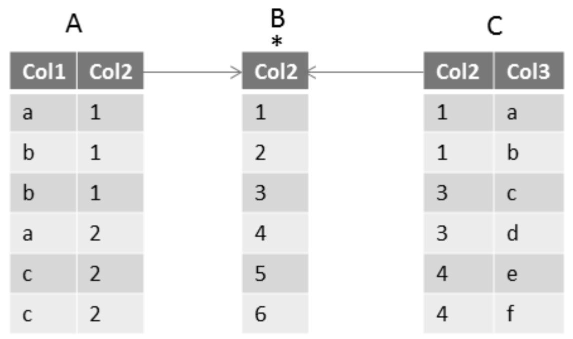
Figure 3.8: Selectivity calculation using Theorem 3.2.3 for another join tree of $R \rightarrow S \rightarrow T \leftarrow M \rightarrow N \leftarrow P \rightarrow Q$

In this report, several theorems in Section 4.2 and Section 4.3 will use Theorem 3.2.3. But in future work, we need to refine this theorem.

We also have some other interesting and useful theorems.

Lemma 3.2.4. *If there is no relation that is referenced by more than one relation, then there is one and only one referencing terminal relation.*

Proof. Assume there are two referencing terminal relations. As the relations are connected, there must be a path between the two referencing terminal relations. Since the two referencing terminal relations are pointing to each other, there must be some relation on the path that is referenced by more than one relation. Contradiction! So there must be no more than one

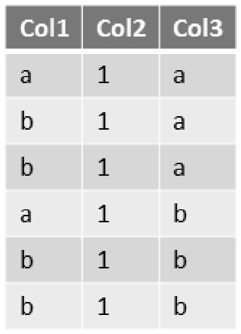


Col1	Col2
a	1
b	1
b	1
a	2
c	2
c	2

Col2
1
2
3
4
5
6

Col2	Col3
1	a
1	b
3	c
3	d
4	e
4	f

Figure 3.9: Three tables, with Col2 in table A referencing primary key Col2 in table B, and Col2 in table C referencing Col2 in table B. Col1 in table A is referencing another primary key. * denotes the primary key.



Col1	Col2	Col3
a	1	a
b	1	a
b	1	a
a	1	b
b	1	b
b	1	b

Figure 3.10: The result of joining tables A, B, and C

referencing terminal relation.

We use induction to prove there must be at least one referencing terminal relation.

Induction Basis: For two relations R and S, there is only one topology $R \rightarrow S$, and there is one referencing terminal relation.

Induction Hypothesis: Assume there is at leaset one referencing terminal relation for any n-relation topology.

Induction Step: We will show there is at least one referencing terminal relation for any (n+1)-relation topology. As the topology is non-cyclic, there are leaves inside, which are connecte to only one relation. We pick up one leaf relation called Q. By the induction hypothesis, there is at least one referencing terminal relations in the other n relations. Assume Q is connected to one referencing terminal relation in the other n relations. If we cannot pick out such leaf, which means the leaf we picked is not connecting to any referencing terminal relation in the other n relations, so those relations are also referencing terminal relations for the (n+1)-relation topology. Assume we can pick up such leaf Q, and Q is connected to the only one referencing terminal relation P in the other n relations. If Q is referencing to P, then Q must be a referencing terminal relation. If Q is referenced by P, then P is still a referencing terminal relation. This completes the proof. $\square$

Theorem 3.2.5. *Suppose there is no relation that is referenced by more than one relation. After joining, the primary key on the referencing terminal relation still holds.*

Proof. Induction Basis: The statement is true for 2 relations. If we have $S \to R$, after joining, the tuples in S will not change, thus the primary key of S is still primary key.

Induction Hypothesis: Assume the statement is true for $n, n-1, ..., 2$ relations.

Induction Step: We need to show the theorem is true for $n+1$ relations. Without loss of generality, we consider only the left subtree referencing the right subtree.

For left subtree, the relation referencing to the right subtree must be referencing to a referencing terminal relation. If not, that is the left subtree is referencing to a referenced relation in the right subtree, then that relation is referenced by two relations, contradiction to our assumption. As the referencing terminal relation still preserves the primary key in the right subtree, by the induction hypothesis, the left subtree is referencing to a primary key on the right subtree. Thus the left subtree does not lose any tuple in the join. As the primary key on the referencing terminal relation from the left subtree still holds on the subtree by the induction hypothesis, it still holds after the final join.

And for the right subtree, there is no referencing terminal relation after the final join, as the only referencing terminal relation is now referenced by a relation from the left subtree. $\square$

Theorem 3.2.6. *Suppose there is no relation that is referenced by more than one relation. If the left subtree is referencing the right subtree, then the size of left subtree is the final join size, and vice versa. Besides, the final join size is just the size of some relation.*

Proof. Induction Basis: Assume the claim is true for 2 relations. For two relations $R \to S$, the final size will be r, which satisfies the theorem.

Induction Hypothesis: The theorem is true for $n, n-1, ..., 2$ relations.

Induction Step: We need to prove the theorem is also true for $n+1$ relations. Without loss of generality, we consider only the case where left subtree references right subtree.

As the proof for Theorem 3.2.5 shows, the left subtree is referencing the right subtree on a primary key. Thus in the final join table, all the tuples for the left subtree is still the same. Then the final join table size is the same as the size of left subtree. And by the induction hypothesis, the size of the left subtree is the size of some relation. Then the final join table size is also the size of some relation. $\square$

Theorem 3.2.7. *Suppose we have n relations with topology $R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow \cdots \rightarrow R_n$, then the final intermediate table size must be r_1.*

Proof. Induction Basis: We can easily see it is true for 3 relations, as shown in Figure 3.11 and 3.12.

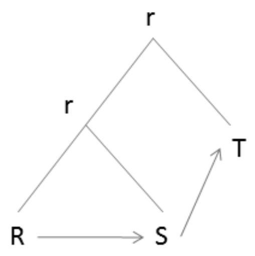


Figure 3.11: 3-relation join tree 1 for topology $R \rightarrow S \rightarrow T$

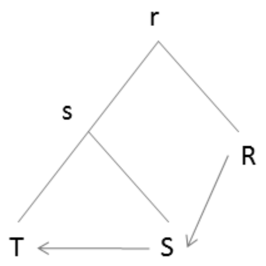


Figure 3.12: 3-relation join tree 2 for topology $R \rightarrow S \rightarrow T$

Induction Hypothesis: Suppose the statement is true for $n, n-1, ..., 3$ relations.

Induction Step: Now we have $n+1$ relations, we know the join tree must have two subtrees and they are divided by a link from where left subtree is referencing right subtree. R_1 must appear on the left subtree, and the final intermediate tuple size for left subtree is thus r_1 by the induction hypothesis. By Theorem 3.2.6, the final join table size is the same as the left subtree, thus is r_1. □

Theorem 3.2.8. *Suppose we have n relations with topology $R_1 \leftarrow \cdots \leftarrow R_{i-1} \leftarrow R_i \rightarrow R_{i+1} \cdots \rightarrow R_n$, then the final intermediate table size must be r_i.*

Proof. Induction Basis: We can easily see it is true for 3 relations, as shown in Figure 3.13 and 3.14.

Induction Hypothesis: Suppose it is true for $n, n-1, ..., 3$ relations.

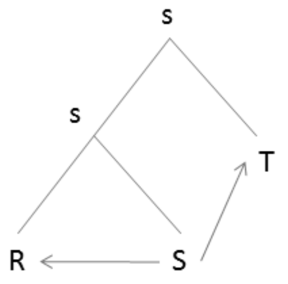


Figure 3.13: 3-relation join tree 1 for topology $R \leftarrow S \rightarrow T$

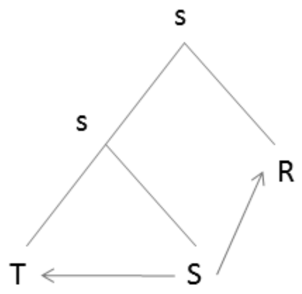


Figure 3.14: 3-relation join tree 2 for topology $R \leftarrow S \rightarrow T$

Induction Step: Now we have $n + 1$ relations, R_i either appears on the left subtree, or the right subtree. If R_i appears on the left subtree, the final intermediate table size for left subtree is thus r_i, by the induction hypothesis. At the same time, as R_i is on the left subtree, the link between left subtree and right subtree is referencing from left to right, and there is no relation that is referenced by more than one relation, then by Theorem 3.2.6, the final join table size is just the size of left subtree, that is R_i.

If R_i appears on the right subtree, the situation is just symmetric to the first case, shown in figure 3.15, and the result follows. $\square$

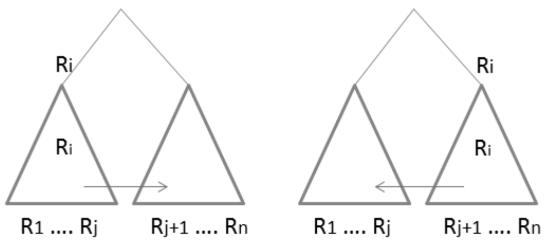


Figure 3.15: The two situations for position of R_i

3.3 Comparable Cost

We want to choose the optimal join tree for a given join query. The query has all the relations involved, and we can build join trees from those relations and estimate their cost using the Theorem 3.2.3. The cost of a join tree is the cost of every join operation, plus the cost of the aggregation operations. But as we are now only concerned about finding optimal join tree, we ignore the aggregation first, thus our join trees only have join operations.

For example, we have two join trees in Figure 3.11 and Figure 3.12. In Figure 3.11, the estimated join tree cost is $3(r+s)+3(r+t) = 3r+3(s+t+r)$, the first item is the cost for $R \bowtie S$, the second item is for the root join. Then similarly, in Figure 3.12, the estimated cost is $3(t+s)+3(s+r) = 3s+3(s+t+r)$. We can see, when we compare costs of the two join trees, the different items are only $3r$ and $3s$. So when we want to compare the costs of the two join trees, we only need to compare size of relation R and S, which are just the sizes of the intermediate tables. We call it **comparable cost**, which means we only consider the intermediate table sizes when doing comparison.

We then try to prove when comparing cost of every join trees of a join query, we can always use comparable cost.

Theorem 3.3.1. *For any join tree, the cost can be rewriten as a form of* $3(\sum$*(intermediate table sizes))*+ $3(\sum$*(leaf table sizes)).*

Proof. Induction Basis: This form is true for 3 relations. Suppose we have R, S and T, regardless of their topology. We can have only one form of join trees, regardless of the order of the relations, shown in Figure 3.16. The cost is just $3(r_1+r_2)+3(s+r_3) = 3s+3(r_1+r_2+r_3)$.

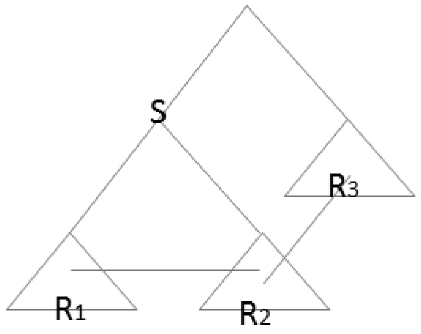


Figure 3.16: The form of 3-relation join tree

Induction Hypothesis: Assume this form is true for join trees of size n, n-1, ..., 3.

Induction Step: For a join tree of n+1 relations, we must have two subtrees, with each one of size smaller than n. So for the two subtrees, their cost can be written as the above form. Now we join the two subtrees, their costs now are the two subtree costs, plus two extra items 3(left subtree table size) and 3(right subtree table size). If the left subtree or right subtree is trivial, which means leaf, the corresponding item is added to the leaf relations sizes part. If neither of them is leaf, then the extra items are both added to the intermediate table size part. In any case, the final join cost is still in the form of 3($\sum$(intermediate table sizes))+ 3($\sum$(leaf relations sizes)). This ends the proof of the join cost form. □

As for any join tree, the cost can be written in this form. Then for the same join query, all the join trees have the same 3($\sum$(leaf relation sizes). When comparing, we only check 3($\sum$(intermediate table sizes)), which means only $\sum$(intermediate table sizes), which is our comparable cost. Then we can just use comparable cost in the computation in the later sections.

We can also have some findings for comparable cost.

Theorem 3.3.2. *In an n-relation join tree, the comparable cost is a sum of $n-2$ items. If there is no relation that are referenced by more than one relation. these items will be the size of some relations.*

Proof. This is because for any full binary tree with n leaves, there must be $n-1$ internal nodes. But we don't count the root table size into comparable cost. Thus the comparable cost, which is the sum of intermediate tuple sizes, have $n-2$ items.

If there is no relation referenced by more than one relation, we know the intermediate table sizes will be just sizes of some relations, by applying Theorem 3.2.6 on all subtrees. □

Theorem 3.3.3. *If there is no relation referenced by more than one relation, the size of referenced terminal relation will not appear in the comparable costs, thus has no effect on the optimality.*

Proof. As there is no relation referenced by more than one relation, then the referenced terminal relations can only be referenced by one relation. Assume we have a referenced terminal relation R, and $S \rightarrow R$. S is inside a subtree Δ in a join tree. Δ cannot join with a subtree that contains R, otherwise, as R has only one link with S, then in that subtree, R is unconnected to other relations, and by Lemma 3.1.1 is joined in a Cartesian product. Therefore Δ joins with R.

By our bottom-up evaluation of join trees, we evaluate Δ and join Δ with R, and then join with other parts, thus the evaluation has three parts.

Inside Δ, the comparable cost cannot contain r, as R is not joined with any relation yet.

As Δ is referencing R, by Theorem 3.2.6, the size of intermediate table of $\Delta \bowtie R$(denote Δ') is the size of Δ.

In the later joins, we are joining Δ' with other relations. As there is no relation referenced by more than one relation, by Theorem 3.3.2, we have all the intermediate table sizes as sizes of some relations. As size of Δ' cannot be r, and in the remaining relations, there is no R, r cannot appear in the rest of intermediate table sizes.

So, in all, the comparable cost can not contain r. The construct is shown in Figure 3.17.

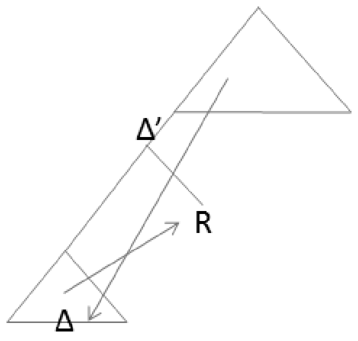


Figure 3.17: The join tree with referenced terminal relation R

If there are any other referenced terminal relation, their sizes also cannot appear in the comparable costs. $\square$

Chapter 4

Optimal Join Tree

The purpose of our study is to find optimal join trees with better performance for RSN schema.

As discussed in Section 1.1, the optimizer of PostgreSQL is using the System R and genetic algorithm. These heuristics are dealing with general cases and may not be very efficient for some specific cases. And the best join tree they find out may not be optimal. In RSN schema, all the join trees are joined on primary key. We use the theories in the previous sections to search for optimal join tree in primary key join queries.

4.1 Linear Join Topology 1

This section looks for features in linear join topology with no relation referenced by more than one relation.

Lemma 4.1.1. *Suppose in a join tree A, we have a subtree called Δ, and its final joined table size is P. The comparable cost for A is a, considering Δ as a relation, and the comparable cost for Δ is d, then the comparable cost for join tree A is $a + d + P$.*

Proof. We can see the Figure 4.1. When we consider Δ as a relation, it is a leaf and the final joined table size will not be counted into the final comparable cost of A. Also the calculation of the comparable cost of Δ does not include the final joined size of Δ. After we plug in the subtree of Δ, the root of Δ is a internal node of A and then the joined size of Δ have to be counted inside.

$\square$

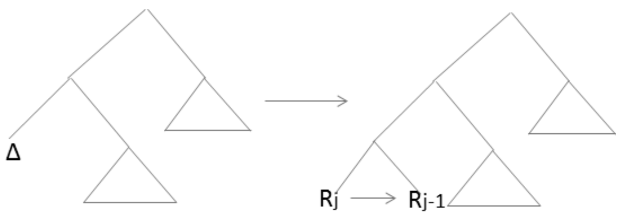


Figure 4.1: The effect of pluggin in a subtree on the comparable cost, need to plus the final joined size of Δ

Theorem 4.1.2. *Suppose we have n relations with topology as $R_1 \to R_2 \to R_3 \to \cdots \to R_n$ ($n \geq 3$). If in $\{R_1, R_2, R_3, ..., R_{n-1}\}$, R_i is the smallest in all the join relations, then the strucutre shown in Figure 4.2 must appear in the optimal join tree.*

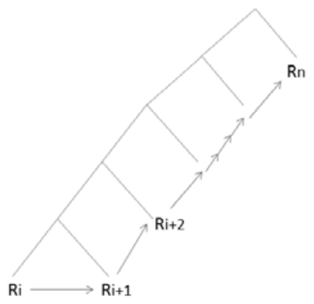
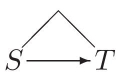


Figure 4.2: The Optimal Strucutre for $R_1 \to \cdots \to R_n$

Proof. We will prove Theorem 4.1.2 by induction.

Induction Basis: The statement is true for $n = 3$. If we have three relations with topology $R \to S \to T$, by Theorem 3.2.1, their join trees are shown in Figure 3.11 and 3.12 with their intermediate join sizes. If $S < R$, then Figure 3.12 is optimal, and this optimal tree contains


$S \longrightarrow T$. If $R < S$, join tree in Figure 3.11 is the optimal, it certainly contains the desired structure, which is itself.

Induction Hypothesis: Assume the statement is true for join trees if the number of relations is $n, n-1, n-2, ..., 3$.

Induction Step: we show this statement is also true for join trees with $n + 1$ relations.

Assume we have a $n+1$ relation join query, with relation topology $R_1 \to R_2 \to R_3 \to \cdots \to R_n \to R_{n+1}$, and R_i has the smallest size in the set $\{R_1, R_2, R_3, ..., R_n\}$. By Theorem 3.1.2 and the position of R_i, we can divide all join trees into 2 types, as shown in Figure 4.3. The type (1) has two subtrees and R_i is in the different subtree as R_{n+1}. The type (2) also has two subtrees, but have R_i in the same subtree as R_{n+1}.

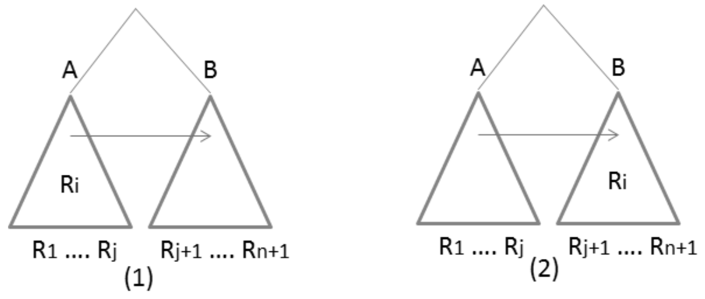


Figure 4.3: The join tree types of $R_1 \to \cdots \to R_n$

In these 2 types of join trees, the right subtree and left subtree both have less than n number of relations. So if we can say only the join trees with subtree containing relations $\{R_i, R_{i+1}, \cdots, R_n, R_{n+1}\}$ can possibly be optimal join tree, then by the induction hypothesis, we can say the optimal join tree must have the desired structure. This means we just need to show the optimal join tree within type (2) is better than that of type (1).

We will try to calculate the tree costs by building up the optimal join tree for both types.

Assume the optimal join tree in type (1) has $\{R_1$, R_2, R_3, ..., $R_k\}$ on subtree A. In subtree A, we have R_i as the smallest relation, then by the induction hypothesis, this part must have structure as in Figure 4.4, we denote it as Δ. Now subtree A has relations $\{R_1$, R_2, R_3, ..., $R_{i-1}\}$, plus the new strucutre Δ. We denote the optimal join tree of subtree B as Γ.

We just treat the subtree Δ and Γ as two relations, then now actually we have a set of relations as $\{R_1$, R_2, R_3, ..., R_{i-1}, Δ, $\Gamma\}$, and we know Γ must be at the top level, then we are just trying to find optimal join tree for relations $D = \{R_1, R_2, R_3, ..., R_{i-1}, \Delta\}$, and the optimal cost is P for D(ignoring subtree cost in Δ). The new formation of the join structure is shown in Figure 4.5.

For part Δ, we know its cost is $(k-i+1-2)r_i = (k-i-1)r_i$, by Theorem 3.3.2. For part Γ, we know it is build by relations $\{R_{k+1}$, R_{k+2}, R_{k+3}, ..., $R_{n+1}\}$, its cost, denoted as Q and $Q > (n-k-1)r_i$ (as Q must be a sum of $(n+1)-(k+1)+1-2 = n-k-1$ items, which in our one-direction linear topology case, are all relation size as shown in Theorem 3.3.2, and each relation sizes are larger than r_i). For part Δ, the final joined size is r_i, by Theorem 3.2.7. For part D, the final intermediate table size must be r_1 by Theorem 3.2.7. Thus, plugging in Δ, D, and Γ, by Lemma 4.1.1, the final comparable cost is

$$P+(j-i-1)r_i+Q+r_i+r_1+r_i > P+(n-j-1)r_i+(j-i)r_i+r_i = P+(n-i+1)r_i \quad (4.1.1)$$

Now let's look at type (2). By the induction hypothesis, type (2) corresponds to join trees that have strucutre shown in Figure 4.6, which means as long as the join tree has the structure

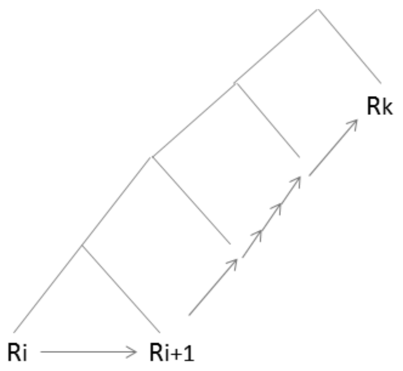


Figure 4.4: Structure of Δ in type (1)

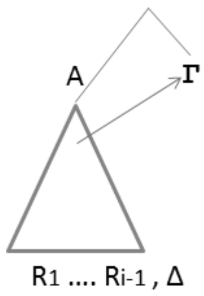


Figure 4.5: Reformed structure for join trees of type (1)

inside, it belongs to type (2). Then we just need to find the optimal tree for join trees that contains the structure in Figure 4.6. We denote the structure in Figure 4.6 as Δ'. For Δ', the final intermediate table size is r_i, by Theorem 3.2.7. Now we have a new set of relations $\{R_1$,

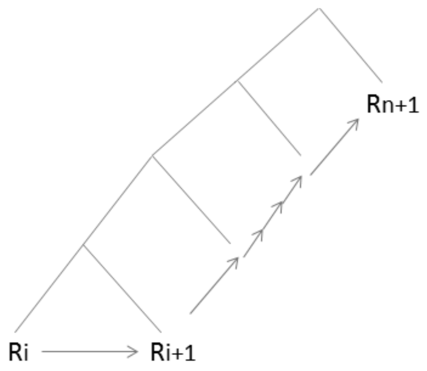


Figure 4.6: Structure Δ' in type (2)

R_2, R_3, ..., R_{i-1}, $\Delta'\}$. This structure is the same as $\{R_1$, R_2, R_3, ..., R_{i-1}, $\Delta\}$, as Δ is the referenced terminal relation, by Theorem 3.3.3. $\{R_1$, R_2, R_3, ..., R_{i-1}, $\Delta'\}$ also has the cost of P(ignoring subtree cost). For part Δ', its cost is $(n+1-i+1-2)r_i = (n-i)r_i$, by Theorem 3.3.2. Thus, plugging in Δ', by Lemma 4.1.1, the final cost is $P+(n-i)r_i+r_i = P+(n-i+1)r_i$.

Thus the optimal join cost for type (2) is smaller(better) than the optimal join cost for type

(1), by Equation 4.1.1. $\square$

We can find another theorem for topology $R_1 \leftarrow R_2 \leftarrow \cdots \leftarrow R_i \rightarrow \cdots \rightarrow R_n$.

Theorem 4.1.3. *Suppose we have n relations with topology as* $R_1 \leftarrow R_2 \leftarrow \cdots \leftarrow R_i \rightarrow \cdots \rightarrow R_n$ *(*$n \geq 3$*). If inside* $\{R_2, R_2, R_3, ..., R_{n-1}\}$*,* R_j *is the smallest, then in the optimal join tree, we can find strucutre shown in Figure 4.7 if* $j \leq i$*, or the structure shown in Figure 4.8 if* $j \geq i$*.*

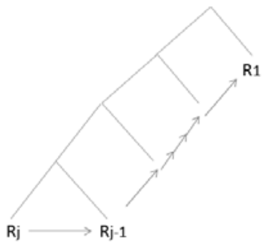


Figure 4.7: The optimal strucutre 1 for $R_1 \leftarrow \cdots \leftarrow R_i \rightarrow \cdots \rightarrow R_n$

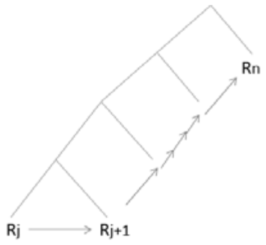


Figure 4.8: The optimal strucutre 2 for $R_1 \leftarrow \cdots \leftarrow R_i \rightarrow \cdots \rightarrow R_n$

Proof. We will prove the statement by induction.

Induction Basis: The statement is true for $n = 3$. If we have three relations with topology $R \leftarrow S \rightarrow T$, by Theorem 3.2.1, their join trees are shown in Figure 3.13 and 3.14 with their intermediate join sizes. Then R_j is R_i. The two join trees are in the same comparable cost, and the optimal tree contains either $\overset{\bigwedge}{S \longrightarrow R}$ or $\overset{\bigwedge}{S \longrightarrow T}$.

Induction Hypothesis: Assume the statement is true for $n, n-1, ..., 3$ relations.

Induction Step: we need to show this statement is also true for join trees with $n+1$ relations.

Assume we have a $(n+1)$-relation join query, with relation topology $R_1 \leftarrow R_2 \leftarrow \cdots \leftarrow R_i \rightarrow \cdots \rightarrow R_{n+1}$, and R_j has the smallest relation size in the set $\{R_2, R_3, ..., R_{n-1}\}$. As we can see, there are two cases for the location of R_j, on the left of R_i and on the right of R_i. For the two cases, the reasoning is symmetric, i.e. in any case, the reasoning is the same, except we need to look towards left or right, so without loss with generality, we can assume R_j is on the left of R_i, that is $j \leq i$.

Since we exclude the Cartesian products, by Theorem 3.1.2, and the location of R_j, we can divide all join trees into 2 types, as shown in Figure 4.9. Type (1) join trees have two subtrees with R_j on the left subtree. Type (2) join trees have R_j on the right subtree.

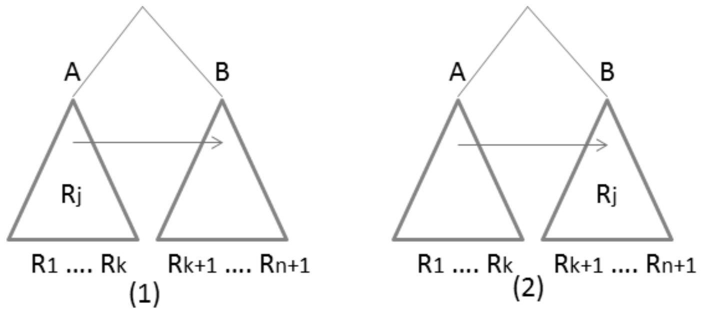


Figure 4.9: The 2 types of join trees for $R_1 \leftarrow R_2 \leftarrow \cdots \leftarrow R_i \rightarrow \cdots \rightarrow R_{n+1}$

In these 2 types of join trees, the right subtree and left subtree both have less than n number of relations. So if we can say only the join trees with its subtree containing relations $\{R_1$, R_2, R_3, ..., $R_j\}$ can possibly be the optimal join tree, then by the induction hypothesis, we can say the optimal join tree must have the desired structure. This means we need to show the optimal join tree within type (1) is better than that of type (2).

We next try to build up the optimal join tree for both types.

A join tree belongs to type (1) as long as R_j is on the left subtree. So, we combine $\{R_1$, R_2, R_3, ..., $R_j\}$, then R_j will be kept on the left subtree. By the induction hypothesis, the left subtree should have structure shown in Figure 4.7, we denote it as Δ, and then we have relations $\{\Delta$, R_{j+1}, R_{j+2}, ..., $R_{n+1}\}$, we construct optimal join tree for these relations, and the optimal join tree cost is P(considering the subtree Δ as a relation). For Δ, its cost is $(j-1+1-2)R_j = (j-2)R_j$, according to Theorem 3.3.2. The final table size for Δ should be r_j. And when we plug in the subtree Δ, the final table size should be counted into the comparable cost, by Lemma 4.1.1. Thus, the cost of the optimal join tree is $P+(j-2)r_j+r_j = P+(j-1)r_j$.

A join tree belongs to type (2) as long as the root left subtree contains at least R_1, and R_j is in the root right subtree. In the optimal join tree, for the root left subtree, assume we have $\{R_1$, R_2, ..., $R_p\}$ for some $p < j$. According to Theorem 3.3.2, and the left subtree topology is $R_1 \leftarrow R_2 \leftarrow \cdots \leftarrow R_p$ as $p < i$, there are $p-2$ items in the optimal cost for the root left subtree and these items are sizes of some relations besides R_1. Say the optimal cost of the left subtree is

Q. As R_j is the smallest of $\{R_2, R_3, ..., R_{n-1}\}$, then $Q > (p-2)r_j$. For the root right subtree, we have $\{R_{p+1}, R_{p+2}, ..., R_{n+1}\}$ and R_j is inside. In order to construct the optimal rightsubtree, we can use the induction hypothesis. As R_j is on the left of R_i, we have the structure shown in Figure 4.10 in the optimal subtree, name it Δ'. Regarding Δ' as a relation, we have in the

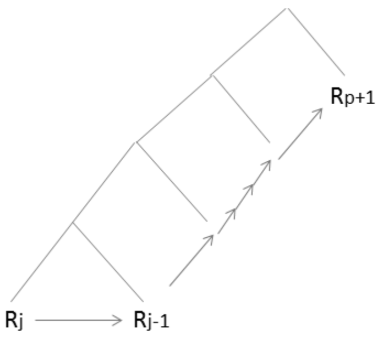


Figure 4.10: Structure Δ' for the root right subtree of type (2)

root right subtree $\{\Delta', R_{j+1}, ..., R_{n+1}\}$. Without considering the cost of subtree Δ', its cost should also be P, by Theorem 3.3.3. The cost of Δ' is $[j-(p+1)+1-2]r_j = (j-p-2)r_j$. The final joined table size for Δ' is r_j, by Theorem 3.2.7. When we plug in Δ' into the right subtree, the comparable cost need to plus the item of final joined table size for Δ', by Lemma 4.1.1. The final joined table size for the left subtree is r_p, by Theorem 3.2.7. The final joined table size for the right subtree is r_i, by Theorem 3.2.8. Plugging in the left subtree and right subtree, by Lemma 4.1.1, the optimal join tree cost is

$$Q+P+(j-p-2)r_j+r_j+r_p+r_i > (p-2)r_j+P+(j-p-2)r_j+r_j+r_j+r_j = P+(j-1)r_j \quad (4.1.2)$$

By Equation 4.1.2, the cost of optimal join tree in type (1) is smaller(better) than that of optimal join tree in type (2). This completes our proof. $\square$

4.2 Linear Join Topology 2

Several of the results in Chapter 3 assumes there is no relation that is referenced by more than one relation(Theorem 3.2.5, Theorem 3.2.6, Theorem 3.3.2 and Theorem 3.3.3). We now consider a case where this assumption is relaxed. Specifically we have a look at the case for having one relation referenced by more than one relation, i.e. $R_1 \to R_2 \to \cdots \to R_i \leftarrow \cdots \leftarrow R_{n-1} \leftarrow R_n$.

By Theorem 3.2.3, for any subtrees, if the values for the referencing terminal relation which connects another subtree is uniformly distributed in that subtree, then we can just apply the selectivity to the join operation of that subtree with another subtree. We will apply that theorem to the join trees in this section.

Theorem 4.2.1. *Assume we have n relations in a linear topology $R_1 \rightarrow R_2 \rightarrow \cdots \rightarrow R_i \leftarrow \cdots \leftarrow R_{n-1} \leftarrow R_n$, with uniform distribution on any joined subtree as shown in the last paragraph.*

Suppose

- *for any integer j and $i+1 \leq j \leq n$, and any integer k, and $i+1 \leq k < j$, we have $r_j < \frac{r_1 r_k}{r_i}$ ...... (*)*
- *for any integer j and $1 \leq j \leq i-1$, and any integer k, and $j < k \leq i-1$, we have $r_j < \frac{r_n r_k}{r_i}$ ...... (**)*

If R_i is smallest in size among all the relations, then the optimal join tree will be in one of the forms shown in Figure 4.11.

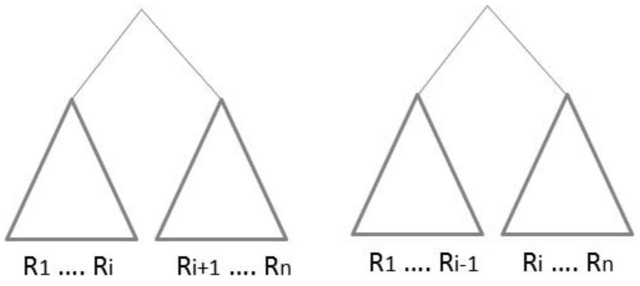


Figure 4.11: Optimal structure for topology $R_1 \rightarrow R_2 \rightarrow \cdots \rightarrow R_i \leftarrow \cdots \leftarrow R_{n-1} \leftarrow R_n$ when R_i is the smallest

Proof. The structures shown in Figure 4.11 is the join trees connected by link $R_i \leftarrow R_{i+1}$ or $R_i \leftarrow R_{i-1}$. To build the optimal join tree, we then apply Theorem 4.1.2 to the two subtrees. We then need to prove, divide the relations from link $R_i \leftarrow R_{i+1}$ or $R_i \leftarrow R_{i-1}$ is better than divide in other links.

Induction Basis: The statement is true for 3 relations. Assume we have three relations in topology $R \rightarrow S \leftarrow T$. There will be only two join trees for this topology, and both of them satisfy the form shown in Figure 4.11. As the optimal join tree is one of them, then the statement is satisfied.

Induction Hypothesis: Assume the statement is true for $n, n-1, ..., 3$ relations.

Induction Step: We need to prove the statement is also true for $n+1$ relations.

Suppose we have $n+1$ relations $R_1 \rightarrow R_2 \rightarrow \cdots \rightarrow R_i \leftarrow \cdots \leftarrow R_n \leftarrow R_{n+1}$. Since we exclude the Cartesian product, by Theorem 3.1.2, and the location of R_i, we can divide all the join trees into 4 types as shown in Figure 4.12. We will prove the optimal join tree in type (1)

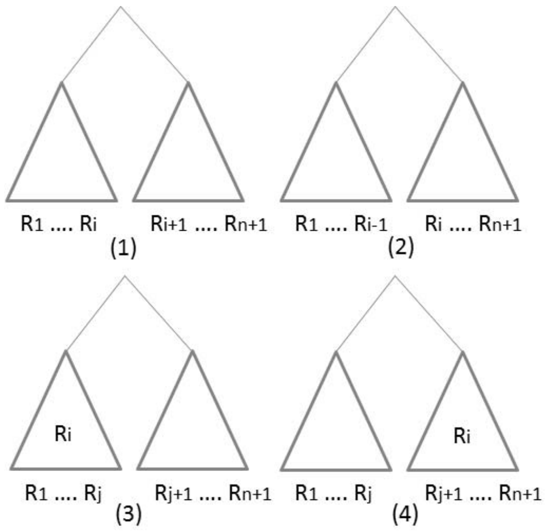


Figure 4.12: 4 types of join trees for topology $R_1 \rightarrow R_2 \rightarrow \cdots \rightarrow R_i \leftarrow \cdots \leftarrow R_n \leftarrow R_{n+1}$

and type(2) is better than that of type (3) and type (4). We will compare type (1) and type (2) with type (3) given condition (*). The comparison of type (1) and type (2) with type (4) is similar under condition (**), and we will not show explicitly.

In type (3), the left subtree has less than n relations, and the topology is $R_1 \rightarrow R_2 \rightarrow \cdots \rightarrow R_i \leftarrow \cdots \leftarrow R_{j-1} \leftarrow R_j$, as (*) is a recursive condition, it is also fulfilled in the subtree, thus we can apply the induction hypothesis to the left subtree, then we have two subtrees for the left subtree, one is formed by relations $\{R_1, ..., R_i\}$ and the other is formed by $\{R_{i+1}, ..., R_j\}$, or one is formed by relations $\{R_1, ..., R_{i-1}\}$ and the other is formed by $\{R_i, ..., R_j\}$.

In the first case, we have type (3) with optimal join tree of the form shown in Figure 4.13. We give the name of subtrees and the intermediate table sizes in the figure. a, p, q are the final joined table sizes for subtrees A, P, and Q respectively. The final joined size of Δ should be $\frac{ap}{r_i}$, by Theorem 3.2.3 and based on our uniformity assumption. Assume the comparable cost for subtree A, subtree P and subtree Q are just A, P, Q correspondingly. Besides, $a = r_1$, $p = r_j$ and $q = r_{n+1}$ by Theorem 3.2.7. Then plugging in A, P and Q, by Lemma 4.1.1, the optimal join tree cost is $A + P + Q + r_1 + r_j + r_{n+1} + \frac{r_1 r_j}{r_i}$.

In the second case, we have type (3) with optimal join tree of the form shown in Figure 4.14. The cost of Δ' should be $\frac{a'p'}{r_i}$, by Theorem 3.2.3 and based on our uniformity assumption.

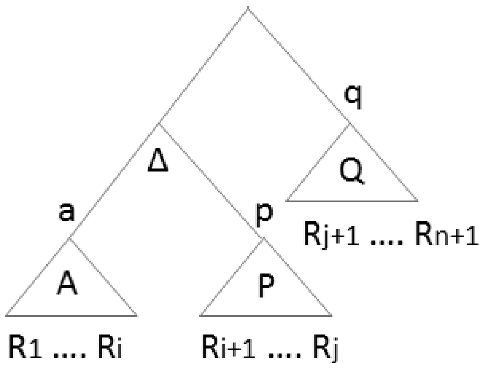


Figure 4.13: The first form of the optimal join tree in type (3)

Assume the comparable cost for subtree A', subtree P' and subtree Q are just A', P', Q correspondingly. In this case, the optimal join tree cost should be $A' + P' + Q + r_1 + r_j + r_{n+1} + \frac{r_1 r_j}{r_i}$.

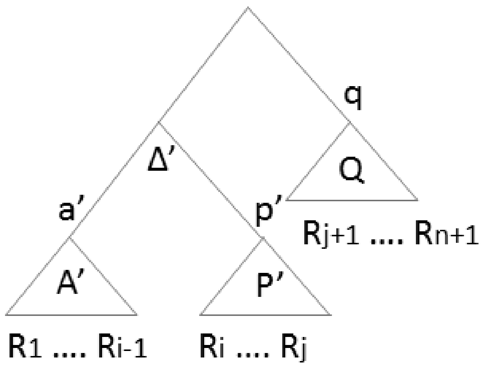


Figure 4.14: The second form of the optimal join tree in type (3)

Now let us have a look at the optimal join tree of type (1). The left subtree is just subtree A in the first case of type (3). The right subtree is the optimal join tree of relations $\{R_{i+1}$, ..., $R_{n+1}\}$, name it D, and assume the optimal cost is D, and the final joined size of r_{n+1}, by Theorem 3.2.7. Then the optimal cost for type (1) is $A + D + r_1 + r_{n+1}$.

Next let us compare D with P and Q. We can join P and Q on the link $R_j \leftarrow R_{j+1}$. and its comparable cost is $P + Q + r_j + r_{n+1}$. As D is the optimal join tree for $\{R_{i+1}$, ..., $R_{n+1}\}$, $P \bowtie Q$ is a join tree for $\{R_{i+1}$, ..., $R_{n+1}\}$, thus $D \leq P + Q + r_j + r_{n+1}$.

Thus for type (1), we have $A + D + r_1 + r_{n+1} \leq A + P + Q + r_1 + r_j + r_{n+1} + r_{n+1} < A + P + Q + r_1 + r_j + r_{n+1} + \frac{r_1 r_j}{r_i}$, by (*).

Comparing type (1) and the first case of type (3), we have the optimal join tree of type (1) better. Similiarly, if we compare type (2) with the second case of type (3), we will have type (2) better.

This completes the proof. □

The searching of the optimal join tree for $R_1 \to R_2 \to \cdots \to R_i \leftarrow \cdots \leftarrow R_{n-1} \leftarrow R_n$ is quite complicated. As we can see from Theorem 4.2.1, only for the case when R_i is smallest, we have to restrict further on the sizes of other relations, then we can confirm the optimal join structure. There are many cases by the size of R_i, we will have complicated case analysis. Or we can try other methods to search for the optimal join tree. Due to the limit of time, we did not have the optimal join tree theorems for all cases of topology $R_1 \to \cdots \to R_i \leftarrow \cdots \leftarrow R_n$. We also need to redefine the selectivity of join operation for the general cases, as discussed after Theorem 3.2.3. The relevant work needs to be done in the future.

4.3 General Linear Topology

Now we come to the general linear topology, for example, $R_1 \leftarrow R_2 \leftarrow R_3 \to R_4 \leftarrow R_5 \to R_6$. We will see how the theorems in Section 4.1 and Section 4.2 can help us.

Picking up any link inside the topology, we can divide the relations into two groups, and each group forms a subtree of some join tree. If on that join tree, we have topology as $R_1 \to R_2 \to \cdots \to R_n$, or $R_1 \leftarrow \cdots \leftarrow R_i \to \cdots \to R_n$, or $R_1 \to \cdots \to R_i \leftarrow \cdots \leftarrow R_n$, then we can apply the corresponding optimal join tree theorems(Theorem 4.1.2, Theorem 4.1.3, Theorem 4.1.3). If not, we continue dividing the subtree, and then search for the known topologies(the above three topologies). For example, for relations $R_1 \leftarrow R_2 \to R_3 \to R_4 \leftarrow R_5$, if we pick link $R_3 \to R_4$, then the left subtree will have $R_1 \leftarrow R_2 \to R_3$, and the right subtree will have $R_4 \leftarrow R_5$. We can apply Theorem 4.1.3 to the left subtree, and Theorem 4.1.2 to the right subtree, which is just linear time. Thus no need to further divide and conquer on the subtrees.

This method of divide and conquer, which in general case, will still be exponential. But in real case, the number of relations in a join query will not be so large, and we might be able to easily find our known topologies. For example, we have 10 relations, and they have topology $R_1 \to R_2 \to R_3 \to R_4 \leftarrow R_5 \leftarrow R_6 \leftarrow R_7 \leftarrow R_8 \leftarrow R_9 \to R_{10}$. There will be 24 join trees to consider. Compared to the case if we don't use our reduction, the number of join trees will be

1430. [1]

In the topology above, there is one relation that is referenced by more than one relation. If there are two, suppose we have $R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_4 \leftarrow R_5 \leftarrow R_6 \leftarrow R_7 \rightarrow R_8 \leftarrow R_9 \rightarrow R_{10}$, where R_4 and R_8 are the relations referenced by more than one relation. In this topology, there will be 392 cases to investigate. Therefore, if there are more than one relation that is referenced by more than one relation, the number of join trees will grow very fast in our approach.

Thus, we have our optimal join tree searching algorithm as follows.

[1]The figure shown in this and next paragraph is calculated using a piece of simple C code. The code can be found at https://dl.dropbox.com/u/6710940/HQ-FYP/code/cases.c

Algorithm 1 Searching for Optimal Join Tree for Primary Key Join Query

```
Require: $n$ relations $\{R_1, ..., R_n\}$, with some linear non-cyclic topology
      SEARCH(topology of $\{R_1, ..., R_n\}$);
  function SEARCH(topology)
      if topology is of the form $R_1 \rightarrow \cdots \rightarrow R_n$ then
          return COST1(topology);
      else if topology is of the form $R_1 \leftarrow \cdots \leftarrow R_i \rightarrow \cdots \rightarrow R_n$ then
          return COST2(topology);
      else if topology is of the form $R_1 \rightarrow \cdots \rightarrow R_i \leftarrow \cdots \leftarrow R_n$ then
          return COST3(topology);
      else
          optimal_cost $\leftarrow \infty$;
          length $\leftarrow$ length(topology)          ▷ length returns the number of links in topology;
          for $i = 1 \rightarrow$ length do
              topo1 $\leftarrow$ topology of $\{R_1, ..., R_i\}$;
              topo2 $\leftarrow$ topology of $\{R_{i+1}, ..., R_{length+1}\}$;
              cost $\leftarrow$ SEARCH(topo1) + SEARCH(topo2) + root_joined_table_size;
              if cost < optimal_cost then
                  optimal_cost $\leftarrow$ cost;
              end if
          end for
          return optimal_cost;
      end if
  end function
  function COST1(topology)
      Apply Theorem 4.1.2 to topology;
      return cost;                          ▷ The cost here includes the root joined table size
  end function
  function COST2(topology)
      Apply Theorem 4.1.3 to topology;
      return cost;                          ▷ The cost here includes the root joined table size
  end function
  function COST3(topology)
      Apply Theorem 4.2.1 to topology;
      return cost;                          ▷ The cost here includes the root joined table size
  end function
```

Chapter 5

Application to PostgreSQL

PostgreSQL uses System R and genetic algorithm in the query optimization. System R, in theory, searches all left-deep and non-Cartesian product join trees and do comparison. However, it suffices from the limit of join tree space, and the inaccuracy of the estimation of join tree cost. Genetic algorithm is a randomized search of join tree which saves search time, but is even worse at finding the optimal join tree. After restricing the join query to primary key join, we have the theorems as stated in the previous chapters, and based on those theorems, we can have an algorithm for faster and better query optimization algorithm.

We are working with PostgreSQL on two aspects: One is to find a way to query on different plans for a query; The second is to do experiments on PostgreSQL so that we can show that query plan we found with the theorems in Chapter 4 is indeed the optimal.

5.1 Query Planning in PostgreSQL

There was some exploration on the possible ways to achieve the first task, for example, using the FYP project from Longwu Huang on Query Specification Language (Huang, 2011), selecting candidate query plans, making use of JOIN clause and so on. These explorations along with the description of the internal of the PostgreSQL system are explained in detail in Appendix A. The use of JOIN clause was tried, but failed in the beginning. The JOIN clause can be used to join two relations, which can be a plain relation or a joined relation. However, the PostgreSQL query rewritter will rewrite the query to a normal join query using just FROM clause. For example, if we want to join three relations R, S, T, we can have these two query plans $(R \bowtie S) \bowtie T$ and $R \bowtie (S \bowtie T)$. If we use just FROM clause to write the joining of these three relations, it will be

```
SELECT * FROM R, S, T WHERE R.r1 = S.s1 AND S.s2 = T.t1;
```

. But if we use JOIN, we can express them differently,

```
SELECT * FROM T JOIN (R JOIN S ON R.r1 = S.s1) ON T.t1 = S.s2;
```

and

```
SELECT * FROM R JOIN (S JOIN T ON S.s2 = T.t1) ON R.r1 = S.s1;
```

. It seems the query 2 and query 3 are different and they are expressing the two query plans mentioned above. But after running them in PostgreSQL, they are exactly the same, if we use EXPLAIN clause to ask PostgreSQL to show the query plan, and they are the same as the result of query 1(EXPLAIN is a PostgreSQL clause that can be used to show the query plan used by the optimizer for the underlying query). This is because the query rewritter just rewrites the JOIN clause to the same form as query 1. But after some search, in PostgreSQL official documentation, it is mentioned that there is an embeded variable *join_collapse_limit* in PostgreSQL that can control the explicit JOIN query (PostgreSQL, 2012a). It is just disabling the rewriting of query using JOIN if the number of relations in the FROM query is more than the nubmer specified by *join_collapse_limit*. Thus we can use command $set\ join\_collapse\_limi = 1$ to disable the rewriting of JOIN query. After using this command, the query 2 and query 3 will behave exactly as it is shown.

Now we are able to run queries as the query plan we want. But we also need to make sure the query plans will only use HashJoin and SeqScan. For each query plan, it will first scan each table based on a scan method, and decide whether to materialize it, and then join the scaned relations using some join methods. We need to disable other scan methods, materialization and other join methods. In file costsize.c, there are variables *enable_seqscan*, *enable_indexscan*, *enable_indexonlyscan*, *enable_bitmapscan*, *enable_tidscan*, *enable_sort*, *enable_hashagg*, *enable_nestloop*, *enable_material*, *enable_mergejoin* and *enable_hashjoin*. They are defaultly set to true. If they are set to $false$, then for the corresponding join method, the optimizer will give the join method an extra cost as specified by variable *disable_cost*, which is default as 1.0^{10}, compared to which most of the join method cost will be much smaller. Then the optimizer will opt out any join tree with this join method. To meet our need, we set all variables to false except for *enable_seqscan* and *enable_hashjoin*.

There is a system wide configuration file postgresql.conf under the ~/bin/data folder. We could configure the value of the above variables inside this file and the values will be consistent in all client sessions.

5.2 Experiments on PostgreSQL

We need to do experiments to show the query plan generated by theorems in Chapter 4 is optimal. The method is to use EXPLAIN to show the optimizer query plan of a query using just FROM clause, then execute the plan and compare the result to the execution of query plans generated by our theorems.

The dataset we use is called ACMDL, which is the data set for ACM digital library [1]. In this report, we focus on the primary key join. We will look at the foreign key constraints in ACMDL. Figure 5.1 shows all the relations in the ACMDL dataset with their attributes that are either foreign keys or primary key.

For the topologies with relations referenced by more than one relation, we did not find a good estimation on the intermediate table size, and there is no good algorithm found for the optimal join plan searching, as stated in Section 3.2 and Section 4.3. We can not do experiment on this kind of topgology yet. In ACMDL, the topology we use that has no relation referenced by more than one relation is $a \leftarrow p \leftarrow pro \rightarrow pe \rightarrow pub$.

Theory shows that the cost of hashjoin is $3(r + s)$, where r and s are the sizes of the two joined relations. In order to make the experiment close to theory, we first need to make the cost of hashjoin close to theory. As the theory on estimation of hashjoin cost only considers disk I/O, we need to make the disk I/O dominant. We scale up the dataset and this will increase the amount of time on disk I/O. We append extra digit behind the primary keys

[1]http://www.acm.org/dl

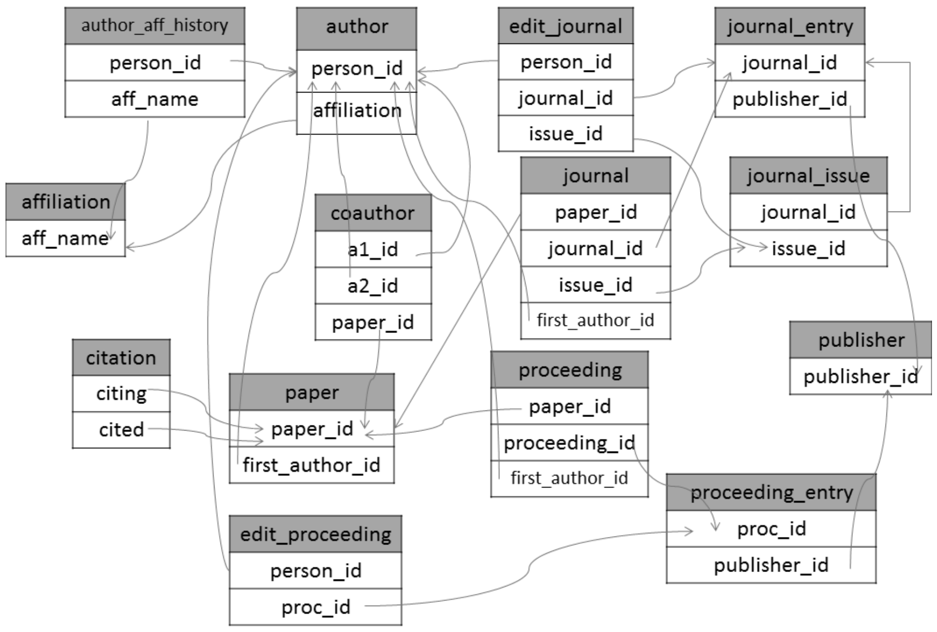


Figure 5.1: The reference relationship for every foreign key in ACMDL. The arrow end denotes the primary key. The arrow start denotes the foreign key.

and foreign keys in the relavant relations. The dataset is upsized 30 times. The relations we scaled up are *author*(1242550 rows, after scaling up and the same below), *paper*(6528960 rows), *proceeding*(4995570 rows), *proceeding_entry*(125280 rows), *publisher*(1200 rows). At the same time, $shared\_buffer$ and $work\_mem$ in PostgreSQL are two variables controlling the shared memory in the hosting system to be used in the execution. We tuned the system with the two variables set to 128kB and 64kB respectively, in order to limit some PostgreSQL execution optimization and encourage more disk I/Os. Also, we discard some large columns in some relations, so that the row width in different relations are about the same.

Another thing to notice about PostgreSQL is that it does not strictly follow System R algorithm, but has some improvement. PostgreSQL considers bushy trees, but only the ones without Cartesian product. As PostgreSQL uses System R algorithm for only small number of relations, thus the number of bushy trees is still not a disaster. If the number of relations is greater than *geqo_threshold*, it will use genetic algorithm which randomizes the optimal join tree selection, which may result in very bad query plan.

The first query we tried is

```
SELECT *
```

```
FROM author1010 a, paper1010 p, proceeding1010 pro, proceeding_entry1010
    pe, publisher1010 pub
WHERE a.person_id = p.first_author_id
AND p.paper_id = pro.paper_id
AND pro.proceeding_id = pe.proc_id
AND pe.publisher_id = pub.publisher_id;
```

. The topology of this query is $a \leftarrow p \leftarrow pro \rightarrow pe \rightarrow pub$. Postgres chooses query plan $((pub \bowtie pe) \bowtie (pro \bowtie p) \bowtie a)$ to execute, and the theoretical optimal query plan is $(pub \bowtie pe) \bowtie ((pro \bowtie p) \bowtie a)$. Howevery, the two queries are of the same comparable cost by theory, which is $(pe + 2 \times pro) = 125280 + 2 \times 4995570 = 10116420$. After executing them, the Postgres query plan runs 239 s, 223 s and 240s. And the theoretical query plan runs 241 s, 240 s and 238 s. So the experiment results are also about the same. Thus we tried another query, which by theory, is worse than the calculated optimal join plan, which is $(a \bowtie p) \bowtie ((pro \bowtie pe) \bowtie pub)$ and its comparable cost is $p + 2 \times pro = 6528960 + 2 \times 4995570 = 16520100$. This plan runs 257 s, 248 s and 260 s.

The second query is

```
SELECT *
FROM paper1010 p, proceeding1010 pro, proceeding_entry1010 pe,
    publisher1010 pub
WHERE p.paper_id = pro.paper_id
AND pro.proceeding_id = pe.proc_id
AND pe.publisher_id = pub.publisher_id;
```

. This query is a subquery of the previous one. The theoretical optimal join plan is $((p \bowtie pro) \bowtie (pe \bowtie pub))$, or $((pe \bowtie pub) \bowtie pro) \bowtie p$. PostgreSQL choose the first join plan as its executing plan.

The third query is

```
SELECT *
FROM author1010 a, paper1010 p, proceeding1010 pro, proceeding_entry1010
    pe
WHERE a.person_id = p.first_author_id
AND p.paper_id = pro.paper_id
AND pro.proceeding_id = pe.proc_id;
```

. The theoretical optimal join plan is $((pro \bowtie p) \bowtie a) \bowtie pe$. The PostgreSQL plan is again the same as the theoretical optimal.

The experiment is covering only few cases, as of the reasons mentioned above. The gain of theoretical optimal join plan over other plans is not so large. This might be because the dataset is still small, and all the different join plans have small difference in cost. The noise still will cause different cost of different runs of the same query, but the affect is much less than before. We also lack the experiments to verify the comparable cost estimation of join plans. According to comparable cost, we can directly tell which query tree is better.

We were also expecting to find the join query in which the theoretical optimal plan will be better than PostgreSQL optimal plan. But for now, there is no such query found. One reason is that PostgreSQL System R algorithm is searching for bushy trees. But with larger data size and larger number of relations in a query, we can expect to find better theoretical join plan than PostgreSQL. On the one hand, we can expect it is harder to find the optimal join plan using System R algorithm on larger dataset, because the inaccuracy of estimation of join cost. On the other hand, if the number of relations in the query is larger than *geqo_threshold*, the genetic algorithm will be used by PostgreSQL, and it will be harder for PostgreSQL to find a good query plan. Thus if we have larger dataset and larger number of relations in the query, the theoretical optimal join plan is expected to do much better than PostgreSQL.

Chapter 6

Suggestions for Future Work

We aim at finding better approaches of searching for the optimal join tree for the queries in RSN. This report focuses on the primary key join, but there may be other benefits of RSN that we can leverage on. For primary key join, the study is not complete and has many enhancement needed. People who want to continue doing research on this topic need to think of the estimation of selectivity in general case again, especially need to solve the case when there is relation that is referenced by more than one relation, as described in the section after Theorem 3.2.3. The optimal join tree searching in the linear topology which have one relation referenced by more than one relation also needs a complete investigation, as described in Section 4.2. This report only considers linear non-cyclic topology, and restricts the join trees on hashjoin, and does not consider of selection, projection and aggregation. These aspects can be improved in the future theoretical research.

The future work should try to consider the real case. Especially if we can be based a RSN-mapped schema, it may give more real situation on what the queries and data will be like. Analyzing the real case schema will bring in some difficulty, especially on how to analyze the data, and the schema, but with more information from real case, it might help restrict the conditions for the theorems, and makes the theorems easier to be established. Otherwise, pure theoretical research may be quite hard to proceed.

Chapter 7

What I've Learned

Doing this FYP project is a great opportunity for me to learn about research. There was a lot of exploration on how to make use of RSN schema. I was trying to start from a concept called acyclic database schema, but later didn't find how to apply it to optimal join plan searching. I then did experiments on some queries directly on ACMDL and aimed at finding some pattern on the optimal join trees. However, there is a lot of noise in the experiment results and makes it hard to find a good pattern. When I started the mathematical analysis on the graphs, then the optimality on join plans is clearer. In this process, the research is often stuck. There are many problems and unknowns which needs to be solved in order to proceed. I am getting more accustomed to problems and know better on how to solve them.

I also learned a lot on PostgreSQL. It is a quite complicated relational database system and the internal system is quite mature after being developed for many decades. There are a lot of good coding coventions and design ideas behind the code. Some tools are needed to help explore the code of PostgreSQL, Linux, bison, vim, ctags, cscope and so on. Also I need to talk with the PostgreSQL community on many problems. Discussing with those guys also give me a lot of knowledge. This research gives me opportunity to work more with PostgreSQL.

I also learned a lot from the discussion with Prof Tay. He gives me a lot of ideas when I am feeling lost in the research.

References

Antoshenkov, G., & Ziauddin, M. (1996). Query Processing and Optimization in Oracle Rdb. *VLDB J.*, *5*, 1996, 229–237.

Chaudhuri, S. (1998). An Overview of Query Optimization in Relational Systems. *PODS*, , 1998, 34–43.

Huang, L. (2011). *Query Plan Specification Translation.* https://dl.dropbox.com/u/6710940/HQ-FYP/Query-Plan-Specification-Translation.doc.

Infosthetics (2008). *Facebook Social Network Graph.* http://infosthetics.com/archives/2008/03 /facebook_social_network_graph.html.

Pingdom, R. (2010). *Exploring the Software Behind Facebook, the Worlds Largest Site.* http://royal.pingdom.com/2010/06/18/the-software-behind-facebook/.

PostgreSQL (2012a). *Controlling the Planner with Explicit JOIN Clauses.* http://www.postgresql.org/docs/current/interactive/explicit-joins.html.

PostgreSQL (2012b). *Genetic Query Optimizer.* http://www.postgresql.org/docs/9.2/static/geqo-intro2.html.

PostgreSQL (2012c). *PostgreSQL: About.* http://www.postgresql.org/about/.

PostgreSQL (2012d). *Query Handling as A Complex Optimization Problem.* http://www.postgresql.org/docs/9.2/static/geqo-intro.html.

Scheufele, W., & Moerkotte, G. (1997). On the Complexity of Generating Optimal Plans with Cross Products. *PODS*, , 1997, 238–248.

Shankar, S., Nehme, R. V., Aguilar-Saborit, J., Chung, A., Elhemali, M., Halverson, A., Robinson, E., Subramanian, M. S., DeWitt, D. J., & Galindo-Legaria, C. A. (2012). Query Optimization in Microsoft SQL Server PDW. *SIGMOD Conference*, , 2012, 767–776.

Tao, Y., Zhu, Q., Zuzarte, C., & Lau, W. (2003). Optimizing Large Star-Schema Queries with Snow Flakes via Heuristic-Based Query Rewriting. *CASCON*, , 2003, 43–57.

Tay, Y. C., Bao, Z., & Zhou, J. *A social network is not a graph: a database view.*

Tay, Y. C., Bao, Z., & Zhou, J. (2012). *sonSQL: A Tool for Starting Up an Online Social Network.* Refer to link: http://sonsql-z.comp.nus.edu.sg/.

Appendix A

Postgres Internal

There is a need for us to be able to specify a query plan for PostgreSQL to execute. I will briefly introduce some relevant internal data structures and work flow of PostgreSQL backend.

PostgreSQL source code resides in a folder *src* under the installation directory $\sim$ if it is installed from source code. *include* folder contains all the head files. The PostgreSQL server code is inside $\sim /src/backend$ folder. The entry point for a query parsing, analyzing, planning and executing is *exec_simple_query* in *backend/tcop/postgres.c*.

Parsing starts from a subroutine called *pg_parse_query*. PostgreSQL uses bison to generate the grammar files. The grammar definition file is gram.y, which lies in folder *backend/parser/*. The keywords are defined in file kwlist.h, in folder *include/parser/*. After compiling, bison will convert the gram.y file to gram.c. Then the C compiler will produce a parser from gram.c.

After analyzing and rewriting the query, planning occurs in subroutine *pg_plan_queries*. Now we will have a look at the work flow of the PostgreSQL planner. The planner information is collected from the query and database, and then stored at a data field *root* of type $PlannerInfo$. Inside a *root*, there is a two dimension list of type $RelOptInfo$, called *simple_rel_array*, which will initially be used to store the base relations in its first cell of its first dimension, then the sub-joined relations, or the intermedaite relations, will be added to the list, according to the number of joined base relations in that intermediate relation. For each intermediate relation, they have several join paths, i.e. join plans. They are stored inside *pathlist* which is a list of data of type *Path*, in a data field of type $RelOptInfo$. After some calculation, all the join paths will be stored inside *simple_rel_array*, and the path with the lowest estimated cost will be stored inside *cheapest_total_path*, inside *RelOptInfo*. The flow starts from a subroutine called *make_one_rel* in file *optimizer/path/allpaths.c*. Subroutine *set_base_rel_size* will estimate the sizes of the base relations. Subroutine *set_base_rel_pathlist* will find all paths for scaning the base relations. Then subroutine *make_rel_from_joinlist* will form all join paths and return the *RelOptInfo* corresponding to the final joined relation. Inside *make_rel_from_joinlist*, we can see that after some processing, it decides whether to use genetic algorithm which will then call subroutine *geqo*, or use System R algorithm which is inside subroutine *standard_join_search*.

After planning, the postgres server will create a data structure called *Portal* which is an abstraction which represents the execution state of a running or runnable query. Subroutine *PortalDefineQuery* will initialize the data fields for queries inside the portal. Then subroutine *PortalRun* will execute the query and then discard the portal by calling *PortalDrop*.

In the beginning, I was planning to use the query plan specification language developed by Longwu Huang (Huang, 2011). This language is an extensive SQL language, and enables the user to specify the join order, scan method and join method. However, when I do test on ACMDL, the result returned by this language is not correct. I was trying to fix it but can not finish in a short time.

Then I hacked into the PostgreSQL in another way. Subroutine *standard_join_search* will search for any non-Cartesian product join plans and store them in *simple_rel_array* of the *RelOptInfo* structure of the final join query. If so, then I can print out all the join paths and find which one is the one I need and point *cheapest_total_path* to that query path, Then the executor will run this query plan. Thus I modified the grammar file to add an optinal index number to SELECT clause. If the index is not specified in a SELECT query, all the found query paths will be printed out in the server log. And if the index is specified, the server will ask planner to point to the selected path to the final join relation path in *simple_rel_array*. As in different run, the order of paths in *simple_rel_array* will be the same, so this modification could work. However, this method is still quite troublesome and is not feasible for experiments with large number of queries.

Appendix B

Resources

The report is attached with a CD which contains some resources.

Queries folder contains some files of queries on the ACMDL. File $10Times_Scale_UP$ is to scale up the original ACMDL dataset to 10 times. File $Scale_UP_30Times$ is scale up the 10 times relations to 3 times of themselves, thus scale the original relations to 30 times.

Patch file *implement.patch* is the patch file for PostgreSQL. It is based on PostgreSQL git repository at commit e40bddb0f3ce9f9bf7c960b35f6ac0c19f65612b. You may checkout the PostgreSQL git main repo at https://github.com/postgres/postgres. It is the code for the implementation mentioned in the last paragraph of Appendix A.

File *postgresql.conf* contains the necessary configuration in order to conduct experiment on PostgreSQL. Just copy the content and paste in the file bin/data/postgresql.conf in your installation folder of PostgreSQL.