PostgreSQL database performance optimization

Qiang Wang

Bachelor Thesis

Business information Technology

2011

HAAGA-HELIA
University of Applied Sciences

Business Information Technology

Abstract

12.04.2011

Authors Qiang Wang	**Group** **X**
The title of your thesis PostgreSQL database performance optimization	**Number of pages and appendices** 47 + 7
Supervisors Martti Laiho	

The thesis was request by Marlevo software Oy for a general description of the PostgreSQL database and its performance optimization technics. Its purpose was to help new PostgreSQL users to quickly understand the system and to assist DBAs to improve the database performance.

The thesis was divided into two parts. The first part described PostgreSQL database optimization technics in theory. In additional popular tools were also introduced. This part was based on PostgreSQL documentation, relevant books and other online resources. The second part demonstrates the optimization process in practice with 3 test cases. Each case was created from a different aspect.

The thesis concludes PostgreSQL database optimization is different from others. Users should have enough knowledge about it before performing database optimization tasks. Especially for those who come from SQL server and DB2 world.

Key words

Indexing, Database, PostgreSQL, Performance, Open source, Configuation

Table of contents

Abbreviation

DDL:	Data Definition Language or Data Description Language
RAM:	random-access memory
DBA:	database administrator
WAL:	write ahead logging (a standard approach to transaction logging)
GiST:	Generalized Search Tree
QUBE:	Quick upper-bound estimate
LRT:	Local response time
TR:	number of random touches
TS:	number of sequence touches
NF:	number of fetch calls
MVCC:	Multiversion concurrency control

1. Introduction

Nowadays software enables business process ever faster and simpler. People are saving either their business data or personal data in electronic form, which leads binary information increased significantly year by year. As one of the most popular data storage, the database management system development has been developed by various companies and communities. There is variety of database management systems: commercials or non-commercials, relational or non-relational and etc. Since the hardware cost and bandwidth is not any more the bottleneck of data processing, database performance optimization bears a considerable amount of attentions from database administrators.

PostgreSQL database management system (later on will use Postgres instead) which was first released at year 1995, is the world's most advanced open source database. A notable amount of application is running on top of it. 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 (PostgreSQL 2010a). Compared with the other open source database system, the extensible functionalities and capabilities of Postgres attracted great interests from DBAs and software developers. And even further, with GNU General Public License, one could not only use the database system free, but also modify and redistribute the database system in no matter what kind of format.

1.1 Background and research problem

Marlevo HR business software is designed and used mainly by small and medium sized company for human resource and customer relationship management. The software has been under active development and use for the last 6 years. Using Marlevo HR software user can access and manipulate needed information through web browser. Currently the amount of data stored in software's Postgres database became larger and it is growing at a geometrical speed. Investment on extending network bandwidth, increasing memory size and CPU capability kept the service response time to a reasonable level. In order to maintain and even shorten the responding time in the

future without too much investment on hardware, tuning database should be the solution.

The primary goal of this thesis project was to identify necessary actions to perform in order to achieve better Postgres database performance. The thesis is divided into two parts; the first part listed theories in general on how Postgres database should be optimized in order to have better data processing; the second part describes how these optimization strategies can be implemented in practice. The theoretical section is collected from database related books, magazines, Postgres user manuals and documentation. Empirical part contains a serial of test cases which were generated from training systems. These cases reveal the common design and implementation of Marlevo Software, and it could be useful also for other software design and implementation.

1.2 The scope of the work

The research result is meant to apply for Postgres 9.0 which was released at 04.10.2010. Earlier versions may not be compatible with this thesis report. The research was carried out under *nix platforms, so the research result is not meant to apply to other platforms. This thesis will mostly concentrate on Postgres configuration tuning and query optimization.

Hardware, underlying operating system and application side optimization will not be covered in the thesis project. However in order to help readers to have better understanding, some relative concepts or terminologies may also be introduced briefly.

The thesis work was done with data provided by Marlevo Software Oy.

2. What is database performance optimization

A real production environment is formed with a serial of different components, hardware like CPU and memory, software like underlying operating system. Postgres is installed on top of the operating system and communicate with other components of the production environment either directly or indirectly. Every production environment is unique with its configuration; if it is not properly configured the overall performance will be degraded. Within the Postgres, retrieving the same information can have many different alternatives, some work faster but the rest are not. (Mullins 2002, 251) (Microsoft 2010a)

The goal of database performance optimization is to maximize the database throughput and minimize connections to achieve the largest possible throughput. It is neither a one-time nor a one day or two's work. It should be considered throughout the development process. Already from the beginning of software development, significant performance improvements can be achieved by designing the database carefully. (Mullins 2002, 251) (Microsoft 2010a) However how database design affects the performance is not the topic of this thesis project.

Mullins mentioned five factors that affect database performance, they are:

- workload
- resource
- optimization
- contention

Workload consists of online transactions, batch jobs, ad hoc queries, data warehousing analysis, and system commands directed through the system at any given time. The workload can be quite different during the period of a day, week or month. (Mullins 2002, 250-251)

Optimization in relational database is unique from other types of system optimization in that query optimization is primarily accomplished internal to the DBMS. (Mullins 2002, 251)

Contention is the condition where two or more components of the workload are attempting to user a single resource in a conflicting way. As contention increases, throughput decreases. (Mullins 2002, 251)

3. Popular Postgres management tools

Postgres database can be created and managed by a variety of different tools. Generally most users prefer using psql query tool, which provides a complete management command set. But unfortunately the psql query tool is a little user un-friendly and requires some background knowledge of command lines. In this chapter briefly described some of the most popular Postgres management tools including psql query tool. Most of them are free and available without expenses. Aqua data studio comes with an expense, but beside its comprehensive ability of database administration, it also provides some advanced functionality that can hardly found from others.

3.1 pgAdmin III

PgAdmin III is the open source tool for Postgres database (PostgreSQL 2010b). It is a powerful database administration and implementation tool shipped with default Postgres installation. It can be used on both *nix and Windows operating system to manage Postgres database since version 7.3 (pgAdmin 2011).

PgAdmin III offers a variety of features like:

- native Postgres access and powerful query tool with color highlighting
- access to all Postgres objects like tables, views, constraints and etc
- database configuration and routine maintenance task management
- wild range of server-side encoding supporting
- users, group and privileges management

It has a visualized query explainer which users can utilize for reading query execution plans. (PostgreSQL 2010b) (pgAdmin 2011) Example below illustrates a typical case of how users can benefit from using it.

```
SELECT w.rowid, w.name AS worker_name, c.name AS customer_name
FROM worker w LEFT JOIN customer c ON (w.parent=c.rowid)
LIMIT 10;
```

	QUERY PLAN text
1	Limit (cost=0.00..3.36 rows=10 width=75)
2	-> Nested Loop Left Join (cost=0.00..134414.00 rows=400000 width=75)
3	-> Seq Scan on worker w (cost=0.00..13434.00 rows=400000 width=26)
4	-> Index Scan using customer_pkey on customer c (cost=0.00..0.29 rows=1 width=57)
5	Index Cond: (w.parent = c.rowid)

Figure 3.1 text based query explainer

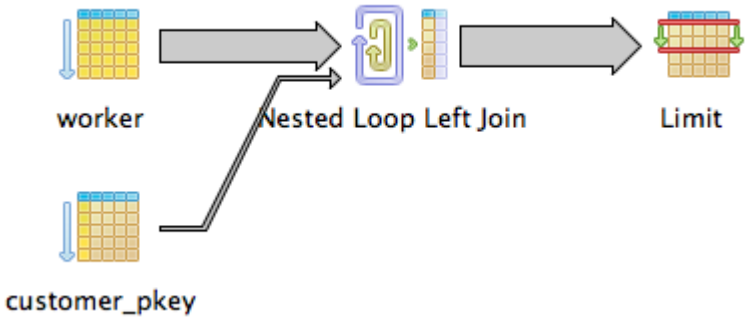

Figure 3.2 visualized query explainer

3.2 phpPgAdmin

phpPgAdmin is a web-based client under GNU General Public License. It is written in PHP programming language and can be installed on a webserver. The browser-based interface it provides is a convenient way for users to perform database administration tasks like:

- manage users and groups
- create database, schema, tablespaces, table, view, sequence and functions
- manage tables, indexes constraints, triggers, rules and procedures
- access table data
- import and export data from / into variety formats

In order to use phpPgAdmin on the webserver, Apache has to be installed first. (PostgreSQL 2010b) (Wikipedia 2010c) (Matthew & Stones 2005, 129-130).

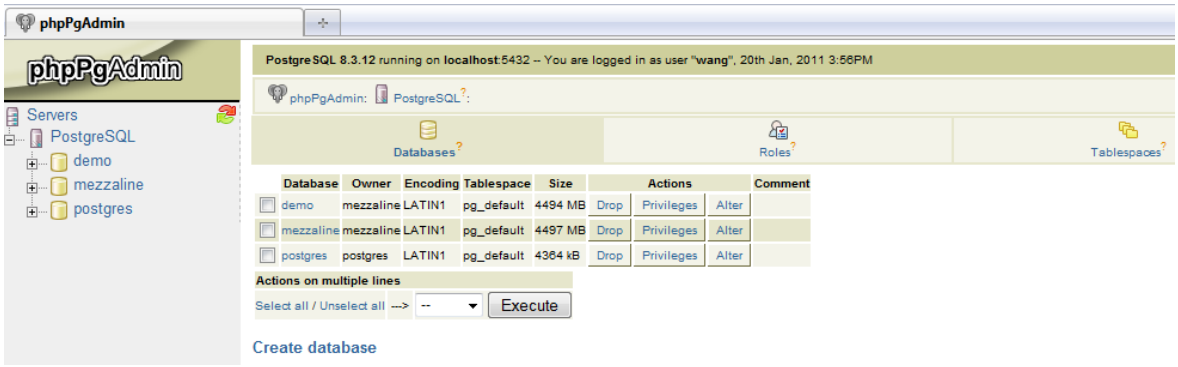

Figure 3.3 phpPgAdmin print screen

3.3 Aqua Data Studio

Aqua Data Studio is a management tool for database query, administration and development. It supports all major relational database management system, like SQL server, DB2, Oracle, MySQL and PostgreSQL. It has been localized for over 20 different languages and can be run on multiple operating systems. With its integrated toolset, users can easily browser and modify database structure, objects, database storage and maintain security. Via Aqua Data Studio, users can easily import and export a database from / into another database management system. Postgres users can use this application to easily create and maintain relational database diagram of their database. However it is a commercial database management tool. The average monthly fee per database user is about 400 US dollar. But a 14 days trial license can be utilized to gain some hand on experience. (PostgreSQL 2010b)

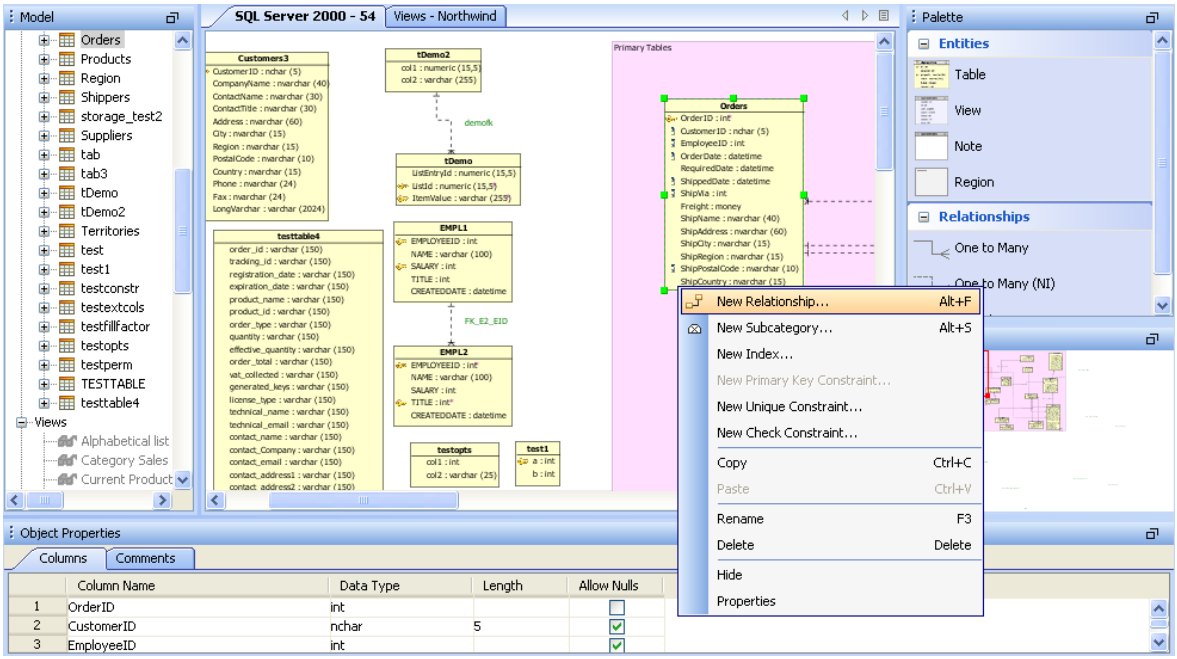

Figure 3.4 creating ER diagram in Aqua Data Studio

3.4 psql command line tool

psql is a terminal-based front-end to Posgres. It is part of the core distribution of Postgres available in all environments and works similarly. Users can issue queries either by manually typing SQL command interactively or executing from a file. Users can perform any kind of database management task via using psql command line tool.

psql commands are categorized into two different types:

- internal psql meta commands
- SQL commands

psql meta commands are used to perform operations which are not directly supported in SQL. These commands are meant to, for example, listing database tables and views, executing external scripts and etc. psql meta commands all start with "\" and cannot be spread across multiple lines. (PostgreSQL 2010c) (Matthew & Stones 2005, 113-115) Running command "\timing" in the command line tool will switch on or off the time counter of query execution. It shows how much time Postgres takes to complete a task.

```
wang-mac:~ wang$ psql -l
                                  List of databases
        Name          |  Owner   | Encoding | Collation | Ctype |   Access privileges
----------------------+----------+----------+-----------+-------+-----------------------
 active_pro_0901      | wang     | LATIN1   | C         | C     |
 active_production    | wang     | LATIN1   | C         | C     |
 marlevo_pro_1311     | wang     | LATIN1   | C         | C     |
 marlevo_pro_2411     | wang     | LATIN1   | C         | C     |
 marlevo_pro_2911     | wang     | LATIN1   | C         | C     |
 marlevo_production   | wang     | LATIN1   | C         | C     |
```

Figure 3.5 List database using psql meta command

```
wang-mac:~ wang$ psql thesis_testing;
psql (9.0.1)
Type "help" for help.

thesis_testing=# \d worker
                                    Table "public.worker"
     Column      |            Type             |                      Modifiers
-----------------+-----------------------------+-----------------------------------------------------
 rowid           | integer                     | not null default nextval('worker_rowid_seq'::regclass)
 parent          | integer                     | not null default 0
 name            | character varying(255)      |
 birthday        | timestamp without time zone |
 email           | character varying(255)      |
 social_security | character varying(255)      |
 contract_start  | timestamp without time zone |
 contract_end    | timestamp without time zone |
 address_street  | character varying(255)      |
 address_city    | character varying(255)      |
 address_zip     | character varying(255)      |
 address_country | character varying(255)      |
Indexes:
    "worker_pkey" PRIMARY KEY, btree (rowid)
    "idx_name_acity" btree (name, address_city)

thesis_testing=#
```

Figure 3.6 Log into database "thesis_testing" and view the structure of table "workers".

```
thesis_testing=# SELECT rowid, name, email FROM worker LIMIT 3;
 rowid |              name               |                email
-------+---------------------------------+---------------------------------------
     2 | e7ZTQK15on6dG1xvcvt0yQgIlz7Pz   | MTVxfqn95ffSfyDpV2
     3 | kLQdpeFMa9QscsRuPV2ZhHwwwWHZ    | DTyi185zmZ0OMUdal3QQqI5B4ya2ElaF
     4 | IW1igCDHQxbjqhSt4ox             | cCjxX2j5pQ4SmUUeBQ
(3 rows)
```

Figure 3.7 Issue an SQL command

4. Performance optimization technics

Performance of a database can be impacted by both internal factors and external factors. Internally, database settings and structure, indexing design and implementation, they all affect the database performance in one way or another. Externally, the physical database design like data file and log file distribution, underlying operating system, they can also form unexpected bottle neck if not well configured. In additional, as more is stored in the database, the possibility of database performance degradation increases. (Mullins 2002, 260)

In Postgres database data accessing can be divided into the following steps:

- transmission of query string to database backend
- parsing of query string
- planning of query to optimize retrieval of data
- retrieval of data from hardware
- transmission of results to client

(Frank)

Factors affects the Postgres database performance and what should be improved in order to retain good database performance were described in detail during this chapter.

4.1 Optimize database configuration

In Postgres configuration settings, there are many parameters affecting the performance of the database system (PostgreSQL 2010c). The default configuration in Postgres is meant for wide compatibility instead of high availability. So if running default installation without tuning necessary parameter, the Postgres database system cannot benefit much from system resource. (PostgreSQL 2011e) Most tunable parameters are in a file called postgresql.conf which is located under Postgres' data directory. Postgres' configuration optimization can be quite a time consuming task and it requires a good understanding of other parts of the system. In this chapter, only the most common tunable parameters will be stated and introduced. Although there are many different ways to configure Postgres' configuration besides editing the

configuration file, here it is means editing the configuration file directly. (Gregory 2010. 125)

4.1.1 max_connections

This number determines the maximum number of concurrent connections allowed in database server. The default setting is normally 100 connections, but could be less depending on the operating system's kernel settings. (PostgreSQL 2010c) Since every time when opening a new connection will cause some overhead to the database and bears a part of shared memories, it is not encouraged to set this number higher than actually needed. If more than a thousand concurrent connections are needed, it is recommended to use connection pooling software to reduce overhead. (Frank) (PostgreSQL 2010c) (PostgreSQL 2011c) One can use the top monitoring tool which is introduced in chapter 4.6 to observe the amount of connections is actually needed by the application.

4.1.2 shared_buffers

This parameter determines how much system memory is allocated to Postgres database for data caching. Having higher shared_buffer value allows more data to be saved in caches and then reduce necessary disk reads (PostgreSQL 2011e). It is the simplest way to improve the performance of a Postgres database server. By default the shared_buffer value is normally set to 32MB which is pretty low for most modern hardware. (Frank) On a system having more than 1 GB memory, it is good practice to allocate 25% of the total RAM, in case of a system having less than 1 GB RAM, allocate 15% out of total RAM is good. (PostgreSQL 2011e)

4.1.3 effective_cache_size

This parameter is used for the query planner to determine how much memory is available for disk caching. It actually does not allocate any memory to the database, but based on this number; the planner will decide whether enough RAM is available if index is used to improve the performance. Normally having this parameter to hold half

of the total RAM is a reasonable setting. More than ¾ of the total memory would lead the query planner to have a wrong estimation. (PostgreSQL 2011e) (Frank) (Gregory 2010, 141)

4.1.4 work_mem

This parameter determines how much memory is allocated to in-memory sort operation. If high values is set, then the query will execute very fast compared with using disk-based sort. By default it is set to be 1 MB. (Gregory 2010, 142) It is a parameter of per-sort rather than per-client. In case a query containing more than one sort operations, the actual memory required will be work_mem times the number of sort operations. (PostgreSQL 2011e) It is difficult to predict how many sort operations are needed per client. One way to calculate suitable value is to see how much free RAM is around after shared_buffers is allocated, divide by max_connections, and then take a part of that figure and then divided by two. (Gregory 2010, 142) Sort operations include ORDER BY, DISTINCT, GROUP BY, UNION AND merge joins (EnterpriseDB 2008, 22).

4.1.5 fsync

With this parameter turning on, Postgres will ensure that data changes will be written back to disks. So that in case of operating system or hardware failure, the database can always return to a consistent state. However it will cause additional overhead while doing WAL. It not recommended of setting this parameter to off, instead turning off synchronous_commit under noncritical circumstance improve the database performance. (PostgreSQL 2010c) (Frank) (Gregory 2010, 143-144)

4.1.6 synchronous_commit

Synchronous commit guarantees a transaction is committed and changes on data are written into disk permanently before it is recorded as complete in WAL. This ensures no data will lose under any circumstances. However, it is known that disk access is quite an expensive operation, having synchronous_commit on will cause significant

overhead in a transaction time. The risk of having this parameter off is the data loss of the most recent transactions before power off or hardware failure. Normally under noncritical environment, it is a good way to have it disabled. (PostgreSQL 2010c) (Gregory 2010, 141-142) (PostgreSQL 2011e)

4.1.7 wal_buffers

This parameter indicates the amount of shared memory allocated for WAL data. The default value is quite small (normally 64 KB). Increasing the default value is helpful for write heavy database system. Up to 16MB is normally a good practice with modern servers. (PostgreSQL 2010c) (Gregory 2010, 138) (PostgreSQL 2011e)

4.1.8 wal_sync_method

This parameter defines the way how Postgres will force WAL updates out to disk. By default the system will use "fdatasync". However it is better to have it set to as "fsync_writethrough" if possible, it is considered to be safe and efficient. (PostgreSQL 2010c) (Gregory 2010, 138) (PostgreSQL 2011e)

4.1.9 random_page_cost

By default in Postgres, the random disk access is considered 4 times slower than sequence disk access. Base on this estimation, the query planner will decide whether it will be faster by using table scan than index scan or vice versa. Depending on the hardware, it is suggested to lower this number when fast disks are use, then query planner will have a higher chance to use index scan. With current hardware, most Postgres database administrators would prefer to have it with value of 2. However, please note in case a query planner is making bad decisions, this should not be the first place to look for answers. How autovacuum works is normally the factor affects the decision of query optimizer. (Frank) (PostgreSQL 2010c) (Gregory 2010, 143) (PostgreSQL 2011e)

4.2 Database table indexing

When database relations are very large, it is very inefficient and expensive to scan all tuples of a relation to find those tuples that match a given condition. In most relational database management system in order to avoid the scenario, indexes are created on tables, it is considered as the single greatest tuning technical that a database administrator can perform. It is helpful in the following scenarios:

- locating a single of a small slice of tuples out of the whole
- multiple table joins via foreign keys
- correlating data across tables
- aggregation data
- sorting data

However, indexing also has negative impacts on performance when data in a relation is inserted or updated more often. During inserting or updating, if there are indexes created on a relation, relative indexes have to be changed as well. So indexes need to be monitored, analyzed, and tuned to optimize data access and to ensure that the negative impact is not greater than positive impact. (Mullins 2002, 298-299) (PostgreSQL 2010c) (Mullins 2002, 261)

Postgres provides several index types: B-tree, Hash, GiST and GIN. Each index type uses a different algorithm that is best suited to different types of queries. One can also build customized index types. (PostgreSQL 2010c) (Gregory 2010, 225)

4.2.1 B-tree index

Balanced tree index is considered as the most efficient index type in Postgres database system. As name implies, the index structure is balanced between its left and right side. Theoretically it requires same travelling distance to any leaf page entry. It can be used to locate a single value or a range of values efficiently if the indexed column is involved in comparison categories of equal, greater than, smaller than or combinations of these operators. B-tree index can also assist in string comparisons such as LIKE, pattern matching if the comparison tries to match the beginning part of the column. However

in case of string comparison, if your database uses other locale other than C, one will need to create index with a special mode for locale sensitive character by character comparison. (PostgreSQL 2010c) (Gregory 2010, 225)

4.2.2 Hash indexes

Hash index can locate records very quickly when only equality comparison is involved in the searching on an index. The hash index entry is organized in key and value pairs, where the value point to the table record(s) contain(s) the key. Given a key, it requires only one or two disk reads to retrieve the table record. It was quite a popular index algorithm in the days when memory was relatively expensive and using B-tree index caused more disk reads. However in Postgres, hash index operations are not WAL-logged, which means it has to be rebuilt manually after system failure. Because of the difficult of maintenance, it is not recommended of implementing Hash index in production system. B-tree index is considered as a good alternative for handling equity comparison. (PostgreSQL 2010c) (Lahdenmäki & Leach 2005, 70) (Ramakrishnan & Gehrke 2003, 279-280) (Gregory 2010, 226)

4.2.3 GiST

B-tree index in Postgres can handle basic data type comparison like numbers and strings using balanced tree structure. In case of customized data type or advanced data type, using the B-tree index could not able to improve the performance. In order to optimize this kind of data types, Generalized Search Tree (GiST) is introduced in Postgres. It is an extensible data structure, which a domain expert can implement for appropriate access method on customized data types or data types that go beyond the usual equality and range comparisons to utilize the tree-structured access method. GiST is also used in full text search by several of contrib modules. (Gregory 2010, 227) (PostgreSQL 2011d) (PostgreSQL 2010a)

4.2.4 GIN indexes

In the regular index structure, a single key is normally associated with either one or multiple values, so finding the desired index entries are relatively simple. But the structure Generalized Inverted Index (GIN) holds is different. For GIN index, a value can have multiple key associated with it. Like GiST, user-defined indexing strategies can be implemented on GIN. It is also a way of implementing full-text search. (Gregory 2010, 226 – 227) (PostgreSQL 2010c)

4.2.5 Cluster operation

In Postgres database, reading records randomly from disk is normally considered 4 times expensive than sequence disk read (Gregory 2010, 240). If records in the database table are randomly distributed, retrieving more than one record will take longer time because of random data access. Clustering a database table using an index will reorganize the order of data records according to the index order, which in turn will reduce the number of random disk access and speed up the process. (PostgreSQL 2010c)

The approach Postgres has taken for clustering is different from the SQL Server and DB2, there is no additional penalty during transactions of having a clustered or clustering index, it does not try to maintain the physical order of records and all new data goes to the end of the table (PostgreSQL 2011c). CLUSTER is not a one-time operation; one has to run CLUSTER every now and then to reorganize the order of records. After the first time of clustering, Postgres will remember which index it was clustered by. During the process of clustering, an exclusive lock is acquired on the table, which will block other transaction from accessing. It is also a time consuming process depending on the data size it holds and the workload of the server. So it is recommended to run during evenings and weekend when others will not be disturbed. Also remember to run ANALYZE after clustering to update the database statistics. (PostgreSQL 2010c) (Gregory 2010, 258)

4.2.6 Bitmap index scan

A single index scan can only use query clauses that use the index's columns with operators of its operator class and are joined with AND. For examples, an index (a, b) is useful on query condition WHERE a=5 AND b=6 but not on query condition WHERE a=5 OR b=6. Fortunately, Postgres has the ability to combine multiple indexes to handle cases that cannot be implemented by single index scan. The system can form AND and OR conditions across several index scans. To combine multiple indexes, the system scan each needed index and prepares a bitmap in memory giving the locations of table rows that are reported as matching that index's conditions. The bitmaps are then ANDed and ORed together as needed by the query. Finally the actual table rows are visited and returned. (PostgreSQL 2010a) (Gregory 2010, 418)

4.2.7 Bitmap heap scan

In Postgres database, data consistency is maintained by using a multi-version model (MVCC). This means when a row is updated by a transaction, the database will create a new version of the row and mark the previous one expired instead of overwriting the old values. By doing this, lock contention is minimized to maintain reasonable performance in multiuser environment. (PostgreSQL 2010c) But it comes with a price. Unlike the Oracle implementation of MVCC that indexes are also versioned, in Postgres, indexes do not have versioning information, therefore all available versions of a row is present in the indexes. Only by looking at the tuple it is possible to determine if it is visible to a transaction. (Dibyendu 2007, 5) So bitmap heap scan means when a slice of an index is acquired by using bitmap index scan, no matter whether the index is fat or not, the database will look at tuples by using the index map to retrieve correct row version.

4.2.8 Join index design

Joining tables together is the most difficult part in the index design. As more tables are joined together, the amount of alternative access paths grows bigger. In case of 3 tables for example, A, B and C joined together, the query planner will have 6 possible access

paths as: A B C, A C B, B A C, B C A, C A B and C B A. Although the final result will be the same, but depending on table access order and join method in use; the elapsed time in total will be quite different. The query optimizer has to evaluate all the possibly access plans and then selects the best one to implement. However evaluating all possible access plans can be quite time consuming in some cases, especially if there are more tables involved. In such cases the query planner will just evaluate a couple of access plans and uses one from them to query the result. This sometime definitely increases the possibilities of choosing the wrong one.

In case the optimizer did not select the best access plan one can add the command "SET join_collapse_limit = 1"in front of the query to be executed. It will limit the query planner to follow the access plan as specified in the query. (Gregory 2010, 262 - 269)

In Postgres when declaring a foreign key constraint to a table an index is not automatically created on the referenced column. In most cases tables are joined together using primary keys and foreign keys, creating index on the foreign key columns will give the optimizer the option to use an index scan instead of table scan if index scan is considered to be faster. (PostgreSQL 2010c) (Lahdenmäki & Leach 2005, 136)

4.2.9 Cost estimation in index design

Before an index is created, one should ask itself questions like how much disk space will be needed to store index entries, whether the index will improve the overall database performance and how the index can help to speed up the database access. (DBTechNet 2010) Query execution cost estimation has been difficult, because each database is running in a unique environment and the elapsed time is closely related to its underlying operating system and hardware. (Lahdenmäki & Leach 2005, 65 - 67)

QUBE (Quick Upper-Bound Estimate) is a cost estimation method introduced by Tapio Lähdenmäki, who worked in IBM Finland. The objective of QUBE is to reveal potential slow access problem during the design phase of a program. Utilizing QUBE

in the index design, one will get estimated elapsed time of a query execution. The formula is:

LRT = TR x 10ms + TS x 0.01ms + F x 0.1ms

In this estimation method, it is assumed that one random disk touch takes 10 milliseconds; one sequence disk touch takes 0.01 milliseconds and transferring one data tuple from disk to memory takes approximately 0.1 milliseconds. The CUBE variables are explained in the following:

- LRT: local response time; how long it takes to execute a query in total
- TR: number of random disk touches; random disk access
- TS: number of sequential touches; sequence disk access
- F: number of successful fetch; how many piece of record to fetch

The QUBE cost estimation method tries to provide the worst scenario for the query execution. Normally in reality, the total elapsed time will be shorter because of the usage of caches. (Lahdenmäki & Leach 2005, 65 - 75)

Sequential touch & random touch

In book "Relational Database Index Design and the optimizers", Lähdenmäki and Leach (2005, 60-70) stated:

> When the DBMS reads one index row or table row the cost is, by definition, one touch: index touch or table touch. If the DBMS scans a slice of an index or table (the rows being read are physically next to each other), reading the first row infers a random touch. Reading the next rows take one sequential touch per row. With current sequential touches are much cheaper than random touches.

Fetch processing

Accepted rows, the number of which will be determined by the number of fetch calls issued, will require a great deal more processing. In QUBE estimate, worst scenario is considered. Normally a row fetch takes 0.1 milliseconds (Lahdenmäki & Leach 2005, 71).

Filter factor

In QUBE estimate, filter factor plays an important role in LRT calculation. According to Lähdenmäki and Leach's statement, the filter factor indicates the selectivity of a predicate, which proportion of the source table rows satisfy the condition expressed by the predicate. So for example, the predicate weekday = 'Monday' will have the average filter factor of 1/ 7 = 14.3 %. And a compound predicate weekday = 'Monday' AND month = 'February', the average filter factor is 1/ 7 * 1 / 12 = 1.2%.

For ORed predicates like weekday = 'Monday' OR month='February', the filter factor calculation will not be straight forward. When predicates of a query are ORed together with other predicates, the predicate may be able to participate in defining an index slice only with multiple index access; otherwise the only feasible alternative would be a full index scan. (Lahdenmäki & Leach 2005, 92 - 93)

4.3 Postgres query explain

As mentioned in previous chapters, when running a query, Postgres can have many alternatives to choose from, some work faster than the others or vice versa and Postgres tries to work out the best one. Also depending on the structure and size of the database, Postgres may not choose the access path as one expected or didn't use the indexes which are meant for it. For the purpose of observing the behavior of Postgres, the built-in utilities of query explain can be used to check the query execution plan and the elapsed time of the query execution in each plan node and in total. Query explain is quite an important and useful tool one can use. (Matthew & Stones 2005, 349 - 350)

The syntax of query explain is like this:
EXPLAIN [(option [, …])] statement
Where options can be one of:

- ANALYZE: the data type is Boolean; it run the append query and show the actual run times, the default value is false, changes on data will write into disk.
- VERBOSE: the data type is Boolean; it displays additional information regarding the plan. Specially; include the output column list for each node in the plan tree, schema-qualify table and function names, always label variables in expression with

their range table alias, and always print the name of each trigger for which statistics are displayed. The default value is false.

- COST: the data type is Boolean; it is used to include information on estimated started up and total cost of each plan node as well as estimation on number of rows and width of each row. The default value for COST is true.
- BUFFERS: the data type is Boolean; it includes information on buffer usage like the number of shared blocks hits, reads and writes, the number of local block hits, reads, and writes, as well as the number of temp blocks reads and writes. Shared blocks, local blocks, and temp blocks contain tables and indexes temporary tables and temporary indexes, and disk blocks contain tables and materialized plans, respectively. This parameter may only be used with ANALYZE parameter. The default value is false.
- FORMAT: specifies the output format. It scan be set as TEXT, XML, JSON or YAML. The default value is TEXT.

(PostgreSQL 2010c)

The example below demonstrate how query explain can be used to check out the query execution plan.

```
EXPLAIN ANALYZE
SELECT rowid, name
FROM customer
WHERE parent=48
ORDER BY name;
```

And the query execution plan is shown as follow:

```
                        QUERY PLAN
-----------------------------------------------------------------------------
 Sort  (cost=1405.63..1407.95 rows=927 width=57)
       (actual time=19.087..19.179 rows=965 loops=1)
   Sort Key: name
   Sort Method:  quicksort  Memory: 115kB
   ->  Bitmap Heap Scan on customer  (cost=47.47..1359.95 rows=927 width=57)
      (actual time=0.638..2.296 rows=965 loops=1)
         Recheck Cond: (parent = 48)
         ->  Bitmap Index Scan on idx_parent_name
             (cost=0.00..47.24 rows=927 width=0)
             (actual time=0.441..0.441 rows=965 loops=1)
               Index Cond: (parent = 48)
 Total runtime: 19.308 ms
(8 rows)
```

Figure 4.1 Explain in command line

When reading the query plan, one should read it from bottom to top:

1. Postgres is doing bitmap index scan on index "idx_parent_name" to get the index bitmap
2. Postgres is doing bitmap heap scan on table customer with the usage of index bitmap from step 1 to retrieve qualified values
3. Postgres sorts the records in the order of customer name. The sort method is quick sort (sort happens in main memory), and memory used is 115 KB.

Visualized query explain is shown in picture below.

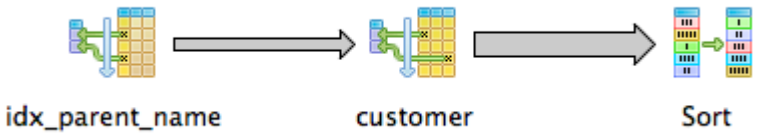

Figure 5.3 Explain in pgAdmin

In figure 5.2, Postgres explain listed figures like costs and actual times in every step. Each of them is used to serve a different purpose:

- Costs: costs are estimations of the time a node is expected to take; each node has both "startup" cost and "total" cost. Cost unit is the cost of a sequential disk page fetch.
- Actual times: actual time of "startup" cost and "total" cost measured in milliseconds.
- Rows: "rows" in cost means the estimated number of records that Postgres will return; "rows" in actual times means how many records Postgres really returns. The difference between the two numbers indicates if Postgres did good estimation or not. If the difference is big, it is most likely because database statistics is out of date.
- Width: how many bytes each record return in current node. This is useful to figure the size of data return in each node.
- Loops: It indicates how many times a node executes. If it has value other than one, then the actual time and row values will be per loop.

(EnterpriseDB 2008, 7-10) (Gregory, S. R239 - 241)

4.4 Postgres procedural language

Stored procedure can be used to implement centralized logic in the database instead of implementing it in the application side. Stored procedures can run directly within the database engine, which makes it have direct access to the data being accessed. It can minimize the round trips between the database and application and at the same time save the processing time. In addition, using stored procedure instead of dynamic SQL will prevent the database from SQL injection. (Wikipedia 2011b)

In Postgres stored procedure can be implemented in a variety of languages including PL/pgSQL, PL/Tcl and etc. (PostgreSQL 2010c). By default PL/pgSQL installed during database installation. In case one prefers to use other language to build stored procedure then the language should be installed first by issuing command CREATELANG (Matthew & Stones 2005, 276-277). Creating stored procedures in PL/pgSQL is quite difficult for beginners with only instructions from Postgres online manual. The book "Beginning Database with PostgreSQL" provides very comprehensive step by step examples on how to utilize stored procedures to the database.

4.5 Prepared Statement Execution

In PHP Manual, Prepared Statements are described as "it can be thought of as a kind of compiled template for the SQL that an application wants to run, that can be customized using variable parameters". It offers two major benefits:

- The query only needs to be parsed once, but can be executed multiple times with the same or different parameters.
- The parameters to prepared statements do not need to be quoted, this avoid SQL injection from occurring. "

Gregory has the same conclusion about Prepared Statement in his book "PostgreSQL 9.0 High Performance". But in addition, he points out that Prepared Statement

Execution could slow down the responds speed in some cases. He says "When a statement is prepared, the query optimizer can only produce a generic query plan for it, but not knowing anything about the actual values that are going to be requested. When you execute a regular query, the statistics about each column available to the optimizer can sometimes do much better than this."

4.6 System monitoring and query logging

Postgres database performance is tied directly to the underlining operating system and how queries are executed. A database administrator needs to know what the database is doing right now and what has caused the degradation of database performance. In this chapter tools were introduced to facilitate system monitoring and query logging

4.6.1 System monitoring

By monitoring the operating system activities, one could know whether the database instance was started; how much resources each process consumes; what the bottleneck of the server is and so on. Here are a couple of popular tools used in *nix operating system:

vmstat

Virtual memory statistics (vmstat) is a system monitoring tool available in most *nix operating system. It displays the most valuable summary about operating system memory, process, interrupts paging and block I/O. (Wikipedia 2011c, Gregory 2010, 326) Figure 4.1 below shows the summary of operating system in 3 seconds' interval, the memory unit is MB.

```
wang@server:~$ vmstat 3 -S M
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
 1  0      0   7863    315    824    0    0     0   263  141   80  0  0 95  4
 0  0      0   7863    315    824    0    0     0    11  547  802 10  1 89  0
 0  0      0   7864    315    824    0    0     0   275  260  211  1  1 96  3
 0  0      0   7864    315    824    0    0     0   263  172  164  0  0 98  2
 1  0      0   7855    315    824    0    0     0    24  416  542  6  1 93  0
```

Figure 4.1 vmstat output

- r: the number of processes that are waiting for run time (Gregory 2010, 327).
- b: the number of processes that are in uninterruptible sleep, if other than 0, system reboot might be needed (Gregory 2010, 327).
- swpd: the amount of virtual memory used, if other than 0, it indicates the system runs out of memory, one should take this into account (Gregory 2010, 81, 327).
- free: the amount of idle memory (Gregory 2010 P327).
- buff: the amount of memory used as buffers (Gregory 2010, 327).
- cache: the amount of memory used as caches (Gregory 2010, 327).
- si / so: the amount of memory swapped in from / to disk (Gregory 2010, 327).
- bi / bo: the blocks received from / sent to a block device (Gregory 2010, 328).
- in: the number of interprets per second (Gregory 2010, 328).
- cs: the number of context switch per second, a dramatic drop of cs indicates the CPU becomes less active due to, for example, waiting for disk drives (Gregory 2010, 328-329).
- us: the percentage of CPU time running external software code, for example Postgres (Gregory 2010, 328).
- sy: the percentage of CPU time running operating system (Gregory 2010, 328).
- id: the percentage of CPU time free (Gregory 2010, 328).
- wa: the percentage of CPU time spent waiting for IO (Gregory 2010, 328).

top

top is an easy tool that can be utilized to get a snapshot of what the system is actually doing right now. It lists the resource usage of each process and together with the overall summary. By default processes are sorted in the order of CPU usage. One can use *-o field* option to sort the process according the *field* value in descending order. (Gregory 2010. 338) (Linux User's Manual. TOP)

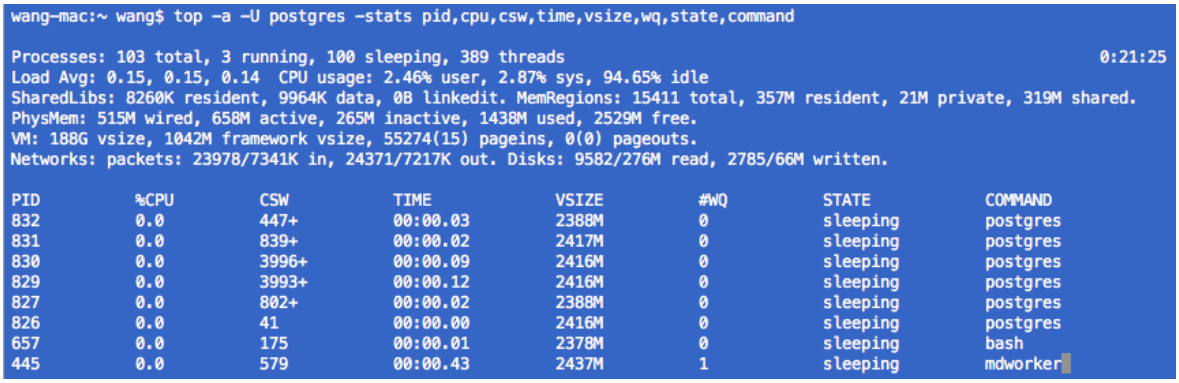

Figure 4.2 top basic outputs

Figure 4.2 shows the overall statistics of the UNIX operating system and summary of the processes which Postgres are running. In this case only selected amounts of statistics were listed:

- PID: the id number of a process (Linux User's Manual. TOP).
- %CPU: the CPU usage of a process (Linux User's Manual. TOP).
- CSW: the number of context switches. It indicates how many thread is running in parallel. (Linux User's Manual. TOP) (The Linux Information Project).
- TIME: the execution time (Linux User's Manual. TOP).
- VSIZE (VIRT in Linux): the total virtual memory used by current process (Linux User's Manual. TOP).
- STATE (S in Linux): the status of current process (Linux User's Manual. TOP).
- COMMAND: command name (Linux User's Manual. TOP)

4.6.2 Query logging

With operating system level monitoring one can get a broad idea about how big the Postgres database server workload is and how it cooperate with the rest parts of the system. However, in order to get detailed information about slow queries and its execution plan database level monitoring should be performed.

This part explains how to collect the queries that slow down server and utilize them to improve performance of Postgres database.

Configure Posgtres to allow query logging

Even though query logging functionality is automatically shipped with Postgres 9.0, it is not enabled by default due to the fact that query logging will cause additional overhead, especially in production environment. In order to enable this functionality some changes should be done in Postgres' configuration file "postgresql.conf". Note query logs here are kept in CSV format, and later on they can be easily imported to database tables for analyzing purpose. (PostgreSQL 2010c, Error Reporting and Logging) (Gregory 2010, 178)

Parameters need to be changed are kept in section "ERROR REPORTING AND LOGGING" in postgresql.conf (short description in bracket):

- log_destincation = 'csvlog' (the format of log records)
- logging_collector = on
- log_directory = 'pg_log' (log file will placed into "pg_log" directory which can be found under Postgres's data directory)

(Gregory 2010, 175-178)

Postgres database has to be restarted in order to make all changes take effect. After that the database will start recording all executed queries into log file. Query logging should be disabled afterwards so that it will not cause any overhead to the database.

Import query log into database table

Create the database table "postgres_log" according to instruction given by Postgres online documentation: http://www.postgresql.org/docs/current/static/runtime-config-logging.html. Then issue the command below to import log file into database.

"COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;"

This will then import query log into database table "postgres_log". (Gregory 2010. 175-179)

5. Index design and performance labs

In the experimental lab, Postgres' EXPLAIN utility was used to observe how queries were executed and its cost distribution. The test cases are to optimize slow and frequently used queries. For the purpose of demonstrating the performance optimization process, 4 database tables were created. And as in most production environment, there were some indexes already created on these database tables. They have some effects while performing query optimization tasks.

The database tables created are:

- customer (stores parent and children customers' information)
- company (stores big customers' information)
- worker (stores workers' information)
- task (keep book of customers and workers' tasks)

The relationship of database tables was shown in figure 5.1. Since the same database is meant for many clients usage, in each of the four tables there is a column called "parent", it is designed as an indicator to identify to which client a record belongs to.

Testing environment and testing data are created as shown below for testing purpose:

- customer: about one hundred thousand records were randomly created with 1% parent entries and 99.99% children entries.
- company: about two hundred thousand records were randomly created
- worker: about four hundred thousand records were randomly created
- task: about eight hundred thousand records were randomly created

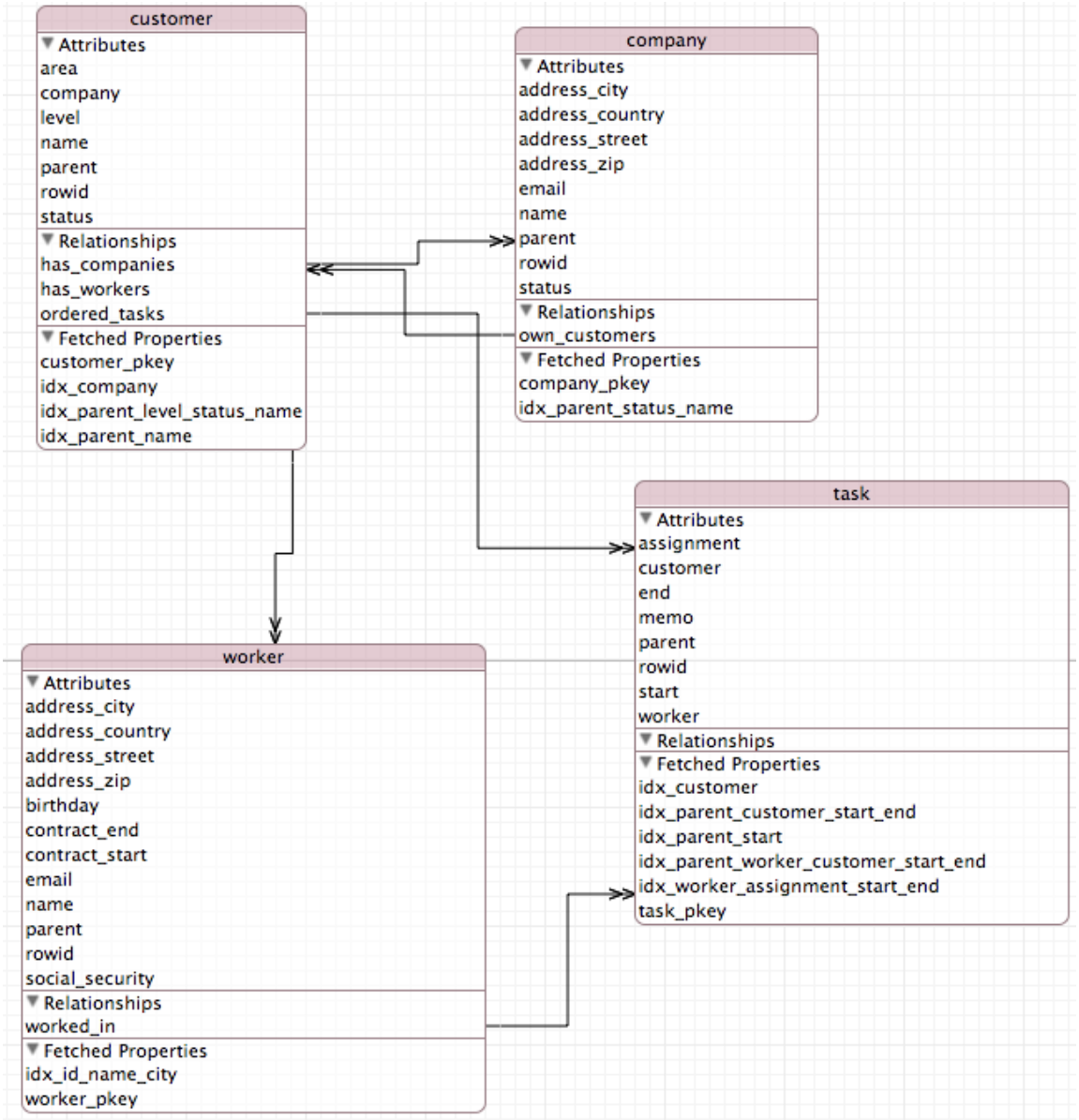

Figure 5.1 database class diagrams

5.1 Clearing cache

Nowadays reading data from disk is still much slower than reading from memories. In order to maintain the overall speed of the system, memory is used to cache the most recent read data, so next time when the same data is requested; the system will read directly from memory instead of disks.

In Postgres, in most cases when execute a query for the second time, the cost of query execution will be much smaller than in the first time. This is called "hot" cache behavior, meaning that the desired data was either stored database cache or in the

operating system caches; the database will read it directly from caches rather than from disks (Gregory 2010, 237-238). But in this experimental lab, the "hot" cache behavior is to be avoided.

Hot caches elimination in both Unix and Linux operating system is presented in below section. It is ought to be performed each time before generating query execution plan.

In UNIX operating system (MAC):

- stop Postgres server

  $ sudo –s

  # su postgres

  # pg_ctl stop –D Postgres/data/directory
- navigate to Users > your user name > Library > Caches and drag all items to Trash
- navigate to Macintosh HD > Library > Caches and drag all items to Trash
- restart
- start Postgres server

  $ sudo –s

  # su postgres

  # pg_ctl start –D Postgres/data/directory

In Linux operating system:

- $ pg_ctl stop
- $ sudo –s
- # sync
- # echo 3 > /proc/sys/vm/drop_caches
- # exit
- $ pg_ctl start

(Gregory 2010, 237-238)

5.2 Test case 1: load customers basic information

This test case demonstrated how different indexes can affect the database performance. The SQL query used in this test case was to querying customers' basic information and lists them in ascending order of customer names.

In the CRM system, customers play quite an important role. Customers' basic information needs to be loaded and listed for selecting when creating events like adding a "meeting customer" calendar entry, creating employment agreement and so forth. The query below is an example which is executed quite often for customers information accessing.

```
SELECT rowid, name FROM customer
WHERE parent= :parent AND level=''
AND (status = '' OR status IS NULL OR status = 'current'
OR status = 'not_delivered')
ORDER BY  name;
```

It can be seen from figure 5.1 that index "idx_parent_level_status_name" seems to fit the SQL query perfectly. But how this index will affect the query performance base on both QUBE theory and query planer?

First here listed the filter factors for each index columns used in this case:
parent: 1 / 10 = 10 %
level: 99 990 / 100 000 = 99.99 %
status: 3 / 7 = 42.9 % (status IS NULL has no entry in testing data)

Then the calculation of the LRT based on QUBE:

Index	parent, level, status, name	TR = 1	TS = 4288
Table	customer	TR = 1	TS = 4288
Fetch	4288 * 0.1 ms		

LRT	TR = 2	TS = 8576
	2 * 10 ms	8576 * 0.01 ms
	20ms + 85.76ms + 428.8ms = 534.56 ms	

The calculation result of 534.56 milliseconds is quite acceptable response time if retrieving 4288 rows from the database. However this estimation is got by using QUBE theory; will the same result be produced in query explain?

```
                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=1959.30..1966.97 rows=3065 width=57) (actual time=129.197..129.495 rows=3345 loops=1)
   Sort Key: name
   Sort Method:  quicksort  Memory: 478kB
   ->  Bitmap Heap Scan on customer  (cost=204.50..1781.82 rows=3065 width=57) (actual time=36.739..68.029 rows=3345 loops=1)
         Recheck Cond: (((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = ''::text)) OR ((parent = 2) AND ((level)::
text = ''::text) AND ((status)::text = 'current'::text)) OR ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = 'not_d
elivered'::text)))
         ->  BitmapOr  (cost=204.50..204.50 rows=3303 width=0) (actual time=30.085..30.085 rows=0 loops=1)
               ->  Bitmap Index Scan on idx_parent_level_status_name  (cost=0.00..4.31 rows=1 width=0) (actual time=18.154..18.154 ro
ws=0 loops=1)
                     Index Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = ''::text))
               ->  Bitmap Index Scan on idx_parent_level_status_name  (cost=0.00..101.14 rows=1667 width=0) (actual time=3.380..3.380
 rows=1682 loops=1)
                     Index Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = 'current'::text))
               ->  Bitmap Index Scan on idx_parent_level_status_name  (cost=0.00..96.75 rows=1636 width=0) (actual time=8.548..8.548
rows=1663 loops=1)
                     Index Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = 'not_delivered'::text))
 Total runtime: 130.243 ms
(13 rows)
```

Figure 5.2 customer data access with index “idx_parent_level_status_name”

In the query planner above, 3345 rows were returned and the actual time spent on retrieving customer rowid - name values pairs is129.495 milliseconds.

The cost in figure 5.2 is as below:

- 3 Bitmap index scan: 30.082 (8.548+3.380 +18.154) milliseconds
- BitmapOr: 0.003 milliseconds
- Bitmap heap scan: 37.944 milliseconds
- Quick sort: 61.466 milliseconds

At first the query planner did 3 times Bitmap index scan on index “idx_parent_level_status_name” to look for valid index entries. Then in step 2, it Ored together all index entries from step 1 to build an index map. In step 3, with the bitmap index, it performed Bitmap heap scan to access table records. Last, quick sort is performed in the memory to sort results in the ascending order of customer name.

In index “idx_parent_level_status_name” the customer name already exists, and customer rowid information is the only information which needs to get from the database table. This probably caused a lot of random disk access and then slows down

the query execution. Most database administrator will consider of creating a fat index by appending rowid to the end of index "idx_parent_level_status_name" to skip the table access. How will it work with Postgres?

```
DROP INDEX idx_parent_level_status_name;
CREATE INDEX idx_parent_level_status_name ON customer
USING btree (parent, level, status, name, rowid);
ANALYZE customer;
VACUUM customer;
```

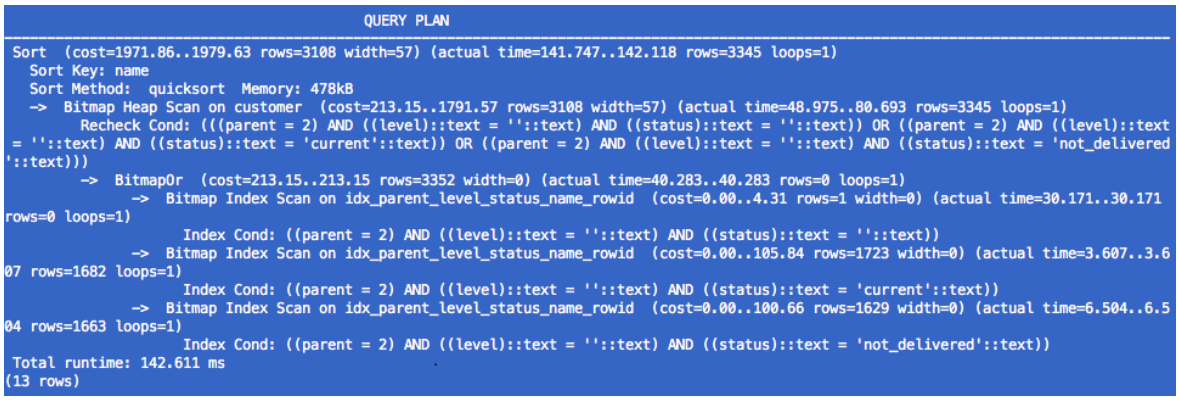

Figure 5.3 data access with fat index "idx_parent_level_status_name_rowid"

With fat index "idx_parent_level_status_name_rowid", the total elapsed time is 142.118 milliseconds, 12.638 milliseconds more than before. The cost is distributed as follow:

- 3 Bitmap index scan: 40.282 (6.504+3.607+30.171) milliseconds
- BitmapOr: 0.001 milliseconds
- Bitmap heap scan: 40.210 milliseconds
- Quick sort: 61.425 milliseconds

By using fat index, the total elapsed time did not become smaller as expected. And the bitmap heap scan was still performed on table customer. How can it happen?

In chapter 4, it is mentioned that Postgres uses MVCC method which is different from most of the other database system for concurrency control. Using this method, in Postgres an index holds all versions of the data, only by consulting data tuple in the table Postgres will know which version is valid. In figure 5.3, even though fat index is

created, it still needs to access the database table to figure out the right version. Appending column rowid to the end of index "idx_parent_level_status_name_rowid" will only make the index size bigger and cannot avoid table access.

In figure 5.2 and 5.3 both of the query planers performed sorting operation in the last steps. It took approximately 60 milliseconds, around 50% of the total elapsed time. For the purpose of reducing sorting cost, next database table customer will be clustered using index "idx_parent_level_status_name".

```
CLUSTER customer USING idx_parent_level_status_name;
ANALYZE customer;
VACUUM customer;
```

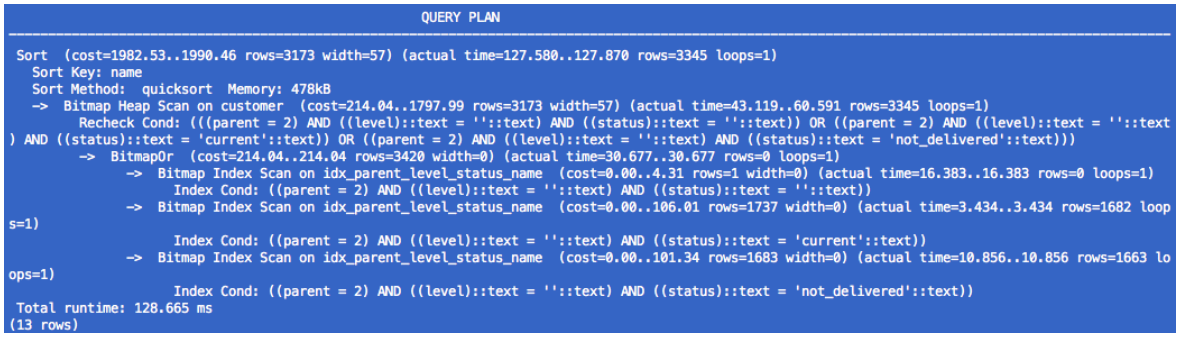

Figure 5.4 customer data access with clustered index

With clustered index on customer, this time the total elapsed time is 127.870 milliseconds. The distribution of the cost is:

- 3 times Bitmap index scan: 30.673 (10.856 + 3.434 + 16.383) milliseconds
- BitmapOr: 0.004 milliseconds
- Bitmap heap scan: 29.914milliseconds
- Sort: 67.279 milliseconds

After clustering the elapsed time of step Bitmap heap scan drop from 37.944 milliseconds to 29.914 milliseconds; however the sorting operation Postgres was 5.813 milliseconds longer. The reason of this is because customer's status is ORed together in the where condition part of the query. Even though customer table is clustered using index "idx_parent_level_status_name", it only reduced the number of random

data access during each Bitmap heap scan node, but the dataset before sorting is still unordered.

OR predicate is a quite difficult predicate for the optimizer. It cannot help in defining the index slice, hence many database administrators use UNION to replace OR. Next step demonstrated how UNION can be different from OR (customer table is still clustered).

```
EXPLAIN ANALYZE
SELECT rowid, name FROM customer
WHERE parent= :parent AND level='' AND status=''
UNION
SELECT rowid, name FROM customer
WHERE parent= :parent AND level='' AND status='current'
UNION
SELECT rowid, name FROM customer
WHERE parent= :parent AND level='' AND
status='not_delivered'
ORDER BY name;
```

```
                                                   QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=3720.81..3729.36 rows=3421 width=57) (actual time=111.770..112.063 rows=3345 loops=1)
   Sort Key: public.customer.name
   Sort Method:  quicksort  Memory: 478kB
   ->  HashAggregate  (cost=3485.78..3519.99 rows=3421 width=57) (actual time=53.492..54.794 rows=3345 loops=1)
         ->  Append  (cost=0.00..3468.68 rows=3421 width=57) (actual time=31.600..48.397 rows=3345 loops=1)
               ->  Index Scan using idx_parent_level_status_name on customer  (cost=0.00..8.32 rows=1 width=57) (ac
tual time=16.161..16.161 rows=0 loops=1)
                     Index Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = ''::text))
               ->  Bitmap Heap Scan on customer  (cost=106.45..1717.77 rows=1737 width=57) (actual time=15.436..18.
992 rows=1682 loops=1)
                     Recheck Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = 'current'::te
xt))
                     ->  Bitmap Index Scan on idx_parent_level_status_name  (cost=0.00..106.01 rows=1737 width=0) (
actual time=3.434..3.434 rows=1682 loops=1)
                           Index Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = 'current'
::text))
               ->  Bitmap Heap Scan on customer  (cost=101.76..1708.37 rows=1683 width=57) (actual time=11.073..12.
532 rows=1663 loops=1)
                     Recheck Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = 'not_delivere
d'::text))
                     ->  Bitmap Index Scan on idx_parent_level_status_name  (cost=0.00..101.34 rows=1683 width=0) (
actual time=1.384..1.384 rows=1663 loops=1)
                           Index Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = 'not_deli
vered'::text))
 Total runtime: 113.546 ms
(16 rows)
```

Figure 5.5 use UNION instead of OR

The cost is distributed as follow:

- 2 times Bitmap index & heap scan and one index scan: 47.685 milliseconds
- Append: 0.712 milliseconds

- HashAggregate: 6.397 milliseconds
- Sort: 57.296 milliseconds

Compared with figure 5.4, it saved about 20 milliseconds by using UNION instead of OR. The elapsed time of sorting operation is also shorter.

5.3 Test case 2: list all tasks ordered by a customer

The previous test case demonstrated how different kinds of indexes affect the performance of single table query execution. In some cases rather than retrieving information from a single table more complicated queries were built to retrieve information across several tables. This test case demonstrated how indexes can be utilized for data selection from several database tables.

The HRM part of Marlevo software is developed for the usage of server different user groups. Main users use it to estimate the workload of their employees; accountants use it to prepare payroll and expense report; and employees use it to check their working schedule. A report contains the following information was frequently used in the system:

- start (from table task; the task start time)
- end (from table task; the task end time)
- assignment (from table task)
- customer name (from table customer)
- worker name (from table worker)
- company name (from table company, if a customer is a branch of a company)

The information is distributed among four tables and their relationship is defined via foreign keys as shown in figure 5.1. Below is a sample query for retrieving the information needed from the database tables. It is a star join and in the where part of the clause there is only one condition t.parent =: parent which determines that table task will be the outmost table.

```sql
SELECT    t.rowid, t.start, t.end, t.memo, t.assignment,
          c.area, c.rowid AS customer_rowid, c.name AS
          customer_name, w.rowid AS worker_rowid, w.name AS
          worker_name, cp.name AS company_name
FROM task t
          LEFT OUTER JOIN worker w ON w.rowid=t.worker
          LEFT OUTER JOIN customer c ON t.customer=c.rowid
          LEFT JOIN company cp ON cp.rowid=c.company
WHERE     t.parent= :parent
ORDER BY  t.start DESC, customer_name;
```

```
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
-----------------------------
 Sort  (cost=99058.68..99264.79 rows=82443 width=304) (actual time=7277.307..7333.985 rows=82240 loops=1)
   Sort Key: t.start, c.name
   Sort Method:  external merge  Disk: 14928kB
   ->  Hash Left Join  (cost=43133.49..69219.75 rows=82443 width=304) (actual time=5751.192..6926.863 rows=82240 loops=1)
         Hash Cond: (t.customer = c.rowid)
         ->  Hash Left Join  (cost=23575.43..41537.83 rows=82443 width=219) (actual time=2798.527..3541.917 rows=82240 loops=1)
               Hash Cond: (t.worker = w.rowid)
               ->  Bitmap Heap Scan on task t  (cost=2303.33..11546.87 rows=82443 width=201) (actual time=1271.818..1403.842 row
s=82240 loops=1)
                     Recheck Cond: (parent = 2)
                     ->  Bitmap Index Scan on idx_parent_start  (cost=0.00..2282.72 rows=82443 width=0) (actual time=1265.540..1
265.540 rows=82240 loops=1)
                           Index Cond: (parent = 2)
               ->  Hash  (cost=13752.60..13752.60 rows=409560 width=22) (actual time=1526.358..1526.358 rows=409560 loops=1)
                     Buckets: 2048  Batches: 32  Memory Usage: 695kB
                     ->  Seq Scan on worker w  (cost=0.00..13752.60 rows=409560 width=22) (actual time=0.009..1197.555 rows=4095
60 loops=1)
         ->  Hash  (cost=16810.06..16810.06 rows=101200 width=89) (actual time=2935.501..2935.501 rows=101200 loops=1)
               Buckets: 1024  Batches: 16  Memory Usage: 781kB
               ->  Merge Left Join  (cost=0.00..16810.06 rows=101200 width=89) (actual time=0.039..2751.415 rows=101200 loops=1)
                     Merge Cond: (c.company = cp.rowid)
                     ->  Index Scan using idx_company on customer c  (cost=0.00..6956.68 rows=101200 width=65) (actual time=0.01
8..2386.164 rows=101200 loops=1)
                     ->  Index Scan using company_pkey on company cp  (cost=0.00..8115.74 rows=189260 width=32) (actual time=0.0
15..170.747 rows=212409 loops=1)
 Total runtime: 7632.268 ms
(21 rows)
```

Figure 5.6 query planner of joins

From figure 5.6 the total elapsed time is 7333.986 milliseconds with 82240 rows returned. The cost distribution is like:

- Bitmap index scan on index “idx_parent_start”: 1265.540 milliseconds
- Bitmap heap scan on task: 138.302 milliseconds
- Sequence scan on worker: 1197.555 milliseconds
- Hash worker: 328.803 milliseconds
- Hash left join task and worker: 611.717 milliseconds
- Index scan on customer: 2386.164 milliseconds
- Index scan on company: 170.747 milliseconds
- Merge left join customer and company: 194.504 milliseconds

- Hash customer and company records: 184.086 milliseconds
- Hash left join task and customer: 449.445 milliseconds
- Sort, external merge: 407.123 milliseconds

Access order: task -> task to worker -> customer to company -> task to customer.

By default in Postgres, if the total amount of available access paths is more than 8, the query planner will evaluate 8 of them and then select the best one to execute. In this case, however, since there are only 6 access paths, the query planner will evaluate all of them and then choose the best one to execute.

From this query planner, the following step cost more time than the others:
- Bitmap index scan on index "idx_parent_start": 1265.540 milliseconds
- Sequence scan on worker: 1197.555 milliseconds
- Index scan on customer: 2386.164 milliseconds

Next the following action was performed to optimize the query execution.

```
CREATE INDEX task_idx_parent ON task USING btree (parent);
CREATE INDEX task_idx_worker ON task USING btree (worker);
CREATE INDEX customer_idx_parent ON customer USING btree
(parent);
CREATE INDEX company_idx_parent ON company USING btree
(parent);
CREATE INDEX worker_idx_parent ON worker USING btree (parent);
CLUSTER customer USING customer_idx_parent;
CLUSTER company USING company_idx_parent;
CLUSTER worker USING worker_idx_parent;
CLUSTER task USING task_idx_parent;
ANALYZE customer;     ANALYZE company;
ANALYZE worker;       ANALYZE task;
VACUUM customer;      VACUUM company;
VACUUM worker;        VACUUM task;
```

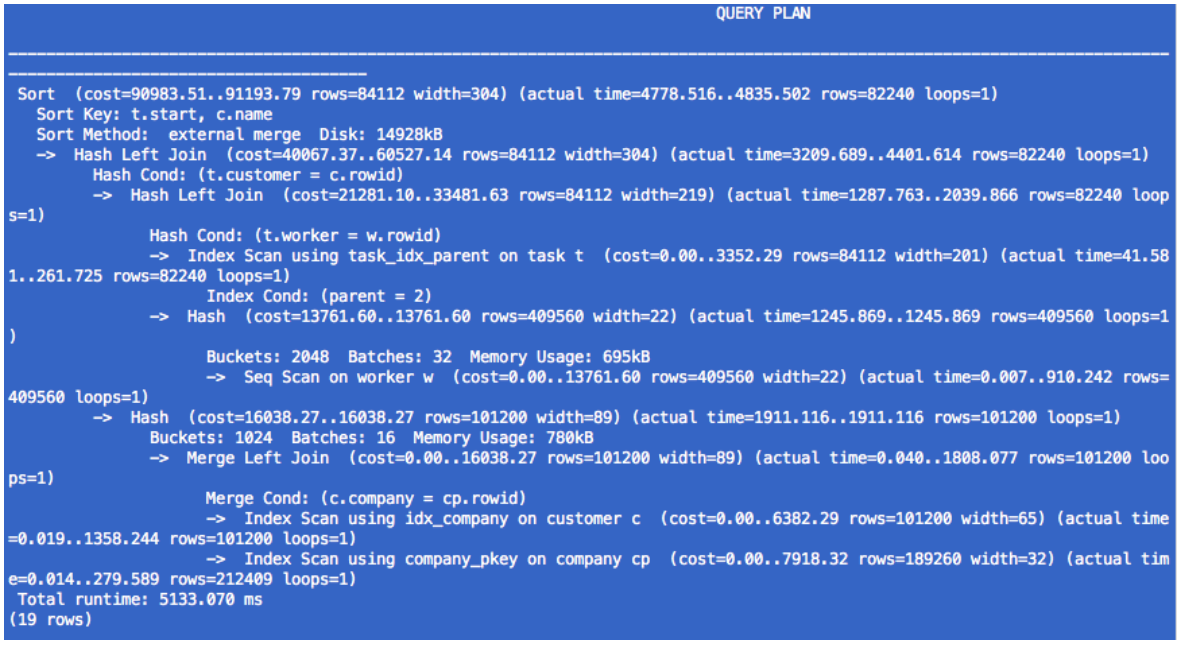

Figure 5.7 query performance after indexing foreign key column and clustering

In figure 5.7, the total elapsed time is 4835.502 milliseconds. The cost distribution is like this

- Index scan on task: 261.725 milliseconds
- Sequence scan on worker: 910.240 milliseconds
- Hash worker: 335.629 milliseconds
- Hash left join task and worker: 532.272 milliseconds
- Index scan on customer: 1358.244 milliseconds
- Index scan on company: 279.589 milliseconds
- Merge left join customer and company: 170.244 milliseconds
- Hash records of customer and company: 103.039 milliseconds
- Hash left join task and customer: 450.632 milliseconds
- Sort, external merge: 433,888 milliseconds

Access order: task -> task to worker -> customer to company -> task to customer.

This time the structure of query execution plan is quite same as in figure 5.6, hence their cost distribution is place in the table for the ease of comparison.

Steps	Figure 5.6	Figure 5.7	Difference
Retrieve task data	1403.842	261,275	1142.567
Retrieve worker data	1197.555	910.242	287.313
Hash worker records	328.803	335.627	-6.824
Join task and worker	611.711	532.272	79.439
Retrieve customer data	2386.164	1358.244	1027.92
Retrieve company data	170.747	279.589	-108.842
Join customer and company	194.504	170.244	24.26
Hash customer and company records	184.086	103.039	81.047
Join task and customer	444.445	450.632	-6.187
Sort	407.123	433.888	-26.765

Figure 5.8 cost comparison of each access node

After adding proper indexes and clustering tables, the cost of data access on task and customer drop dramatically. But still accessing data of inner table worker, customer and company cost long time. According to Figure 5.1, each database table has column "parent" which indicates to which Marlevo client does a record belongs to. Next step more query conditions were added to the where part of the query to limit the return result of inner tables.

```
SELECT    t.rowid, t.start, t.end, t.memo, t.assignment,
c.area AS customer_area, c.rowid AS customer_rowid, c.name
AS customer_name, w.rowid AS worker_rowid, w.name AS
worker_name, cp.name AS company_name
FROM task AS t
LEFT OUTER JOIN worker AS w ON w.rowid = t.worker
LEFT OUTER JOIN customer AS c ON t.customer = c.rowid
LEFT JOIN company AS cp ON cp.rowid = c.company
WHERE t.parent = :parent AND c.parent = :parent
AND cp.parent = :parent AND w.parent = :parent
ORDER BY t.start DESC, customer_name;
```

```
                                                                                QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------
 Sort  (cost=16784.16..16799.08 rows=5968 width=304) (actual time=1861.304..1917.473 rows=82240 loops=1)
   Sort Key: t.start, c.name
   Sort Method:  external merge  Disk: 14944kB
   ->  Hash Join  (cost=5461.51..15573.38 rows=5968 width=304) (actual time=657.700..1512.621 rows=82240 loops=1)
         Hash Cond: (t.worker = w.rowid)
         ->  Hash Join  (cost=2684.10..11419.53 rows=12772 width=286) (actual time=389.111..813.314 rows=82240 loops=1
)
               Hash Cond: (t.customer = c.rowid)
               ->  Index Scan using task_idx_parent on task t  (cost=0.00..3352.29 rows=84112 width=201) (actual time=
49.901..277.108 rows=82240 loops=1)
                     Index Cond: (parent = 2)
               ->  Hash  (cost=2362.17..2362.17 rows=11834 width=89) (actual time=339.096..339.096 rows=12000 loops=1)
                     Buckets: 1024  Batches: 2  Memory Usage: 727kB
                     ->  Hash Join  (cost=1221.93..2362.17 rows=11834 width=89) (actual time=158.802..326.350 rows=120
00 loops=1)
                           Hash Cond: (c.company = cp.rowid)
                           ->  Index Scan using customer_idx_parent on customer c  (cost=0.00..509.36 rows=11834 width
=65) (actual time=13.113..111.298 rows=12000 loops=1)
                                 Index Cond: (parent = 2)
                           ->  Hash  (cost=851.91..851.91 rows=19122 width=32) (actual time=145.494..145.494 rows=1934
0 loops=1)
                                 Buckets: 2048  Batches: 2  Memory Usage: 632kB
                                 ->  Index Scan using company_idx_parent on company cp  (cost=0.00..851.91 rows=19122
width=32) (actual time=18.331..129.902 rows=19340 loops=1)
                                       Index Cond: (parent = 2)
         ->  Hash  (cost=2055.12..2055.12 rows=39304 width=22) (actual time=268.441..268.441 rows=39600 loops=1)
               Buckets: 2048  Batches: 4  Memory Usage: 530kB
               ->  Index Scan using worker_idx_parent on worker w  (cost=0.00..2055.12 rows=39304 width=22) (actual ti
me=10.347..237.208 rows=39600 loops=1)
                     Index Cond: (parent = 2)
 Total runtime: 2148.888 ms
(24 rows)
```

Figure 5.9 Added more query condition in where clause for table joins

This time the total cost of the query execution is 1917.473 milliseconds. Here is the cost distribution:

- Index scan on task: 277.108 milliseconds
- Index scan on customer: 111.298 milliseconds
- Index scan on company: 129.920 milliseconds
- Hash company: 15.574 milliseconds
- Hash join customer and company: 69.558 milliseconds
- Hash records of customer and company: 12.764 milliseconds
- Hash join task and customer: 196.246 milliseconds
- Index scan on worker: 237.208 milliseconds
- Hash: 31.233 milliseconds
- Hash join task and worker: 430.866 milliseconds
- Sort: 404,852 milliseconds

Access path: task -> customer to company -> task to customer -> task to worker.

Now cost in most access nodes have drop quite much. But the last access node of each query execution plan "sort" has not changed too much, it was still around 400 milliseconds.

The sorting method "external merge" in used indicated that the amount of memory allocated to Postgres for sorting operation is not enough. Hence Postgres has to swap records into disk to perform the sort operation. Recall from chapter 4.1.4, sorting operation on disk is normally slower compared to sorting in the main memory. So, raise work_mem in Postgres' configuration to 15 MB will help to reduce the cost future (the value of disk parameter in sort node).

5.4 Test case 3: prepared query execution

Test case 3 demonstrated how prepared query execution is different normal query execution with both hot cache and cold cache. In this test case, query in figure 5.10 will be used.

```
-- 1. PREPARE THE SQL STATEMENT --
PREPARE get_customers (int) AS
SELECT    rowid, name FROM customer
WHERE     parent = $1
          AND level = ''
          AND status = 'not_delivered'
ORDER BY  name;


-- 2. EXECUTE PREPARED SQL STATMENT --
EXECUTE get_customers (2);

-- 3. NORMAL QUERY EXECUTION STATEMENT --
SELECT    rowid, name FROM customer
WHERE     parent = :parent
          AND level = ''
          AND status = 'not_delivered'
ORDER BY  name;
```

The query planner statistics is displayed as below:

```
thesis_testing=# EXPLAIN ANALYZE EXECUTE get_customers (2);
                              QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=1906.74..1910.83 rows=1637 width=57) (actual time=121.910..122.052 rows=1663 loops=1)
   Sort Key: name
   Sort Method:  quicksort  Memory: 238kB
   ->  Bitmap Heap Scan on customer  (cost=221.25..1819.35 rows=1637 width=57)
       (actual time=59.418..91.437 rows=1663 loops=1)
         Recheck Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = 'not_delivered'::text))
         ->  Bitmap Index Scan on idx_parent_level_status_name
             (cost=0.00..220.84 rows=1637 width=0) (actual time=49.000..49.000 rows=1663 loops=1)
               Index Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = 'not_delivered'::text))
 Total runtime: 122.957 ms
(8 rows)
```

Figure 5.10 Prepared query execution with cold cache

```
thesis_testing=# EXPLAIN ANALYZE EXECUTE get_customers (2);
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=1817.58..1821.06 rows=1390 width=57) (actual time=32.608..32.746 rows=1663 loops=1)
   Sort Key: name
   Sort Method:  quicksort  Memory: 238kB
   ->  Bitmap Heap Scan on customer  (cost=186.10..1745.02 rows=1390 width=57) (actual time=1.921..3.694 rows=1663 loops=1)
         Recheck Cond: ((parent = $1) AND ((level)::text = ''::text) AND ((status)::text = 'not_delivered'::text))
         ->  Bitmap Index Scan on idx_parent_level_status_name  (cost=0.00..185.75 rows=1390 width=0)
             (actual time=1.822..1.822 rows=1663 loops=1)
               Index Cond: ((parent = $1) AND ((level)::text = ''::text) AND ((status)::text = 'not_delivered'::text))
 Total runtime: 32.931 ms
(8 rows)
```

Figure 5.11 Prepared query execution with hot cache

Below are the normal query executions:

```
                                                QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=1906.74..1910.83 rows=1637 width=57) (actual time=121.727..121.877 rows=1663 loops=1)
   Sort Key: name
   Sort Method:  quicksort  Memory: 238kB
   ->  Bitmap Heap Scan on customer  (cost=221.25..1819.35 rows=1637 width=57) (actual time=59.403..91.341 rows=1663 loops=1)
         Recheck Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = 'not_delivered'::text))
         ->  Bitmap Index Scan on idx_parent_level_status_name  (cost=0.00..220.84 rows=1637 width=0)
             (actual time=48.883..48.883 rows=1663 loops=1)
               Index Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = 'not_delivered'::text))
 Total runtime: 122.771 ms
(8 rows)
```

Figure 5.12 Normal query execution with cold cache

```
                                                QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=1906.74..1910.83 rows=1637 width=57) (actual time=30.755..30.892 rows=1663 loops=1)
   Sort Key: name
   Sort Method:  quicksort  Memory: 238kB
   ->  Bitmap Heap Scan on customer  (cost=221.25..1819.35 rows=1637 width=57) (actual time=1.510..2.743 rows=1663 loops=1)
         Recheck Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = 'not_delivered'::text))
         ->  Bitmap Index Scan on idx_parent_level_status_name  (cost=0.00..220.84 rows=1637 width=0)
         (actual time=1.449..1.449 rows=1663 loops=1)
               Index Cond: ((parent = 2) AND ((level)::text = ''::text) AND ((status)::text = 'not_delivered'::text))
 Total runtime: 31.060 ms
(8 rows)
```

Figure 5.13 Normal query execution with hot cache

As shown in figures above that the cost difference between prepared query execution and normal query execution was not obvious when running a single query in Postgres database. The prepared query execution takes even a little longer time than normal query execution (a couple of milliseconds, for example). However in production

environment, the result may vary due to the huge amount of queries are running in parallel. To maintenance higher availability and security of the database, it is a good practice to implement prepared query execution.

In attachment 1, one can find the sample code of how to utilize prepared query execution in PHP programming.

6. Conclusion

There is variety of factors affecting Postgres database performance. The thesis covered database performance tuning technics from the aspects of database configuration tuning and query optimization. In addition, some popular management and monitoring tools are introduced in chapters 3 and 4.

Postgres database configuration tuning is the first thing a DBA should do after installation. By default Postgres is configured for wilder compatibility instead of higher availability. Without proper configuration, Postgres can hardly gain much benefit from system resources and it sometimes lead users especially juniors to the conclusion that Postgres is slower than other DBMS.

Performing Postgres database configuration tuning requires some background knowledge on both hardware and underlying operating system. Changes must be properly tested before deploying into a production environment; otherwise the configuration tuning may even harm the overall performance. Users can use system monitoring tool to keep track of their effects.

Query optimization is a different kind of optimization from other. It is implemented within the database system. Most of relational database support some kinds of optimization technics for fast data access. However, even though the idea of query optimization is the same, different database management system has different implementation. The most popular query optimization method is indexing. It is also a main part of this thesis project.

Postgres uses MVCC algorithm to handle multi-users concurrent read and write access to the same data. It is quite different from most other database management system. Users must be aware of this while indexing on queries, especially for those who come from SQL server and DB2 world. In Postgres, it is not encouraged to create fat indexes due to the fact that the data tuple in tables always needs to be accessed to

determine the visibility of index values. Creating multiple small indexes instead of one fat index will normally bring better performance.

In the experimental part of the thesis project, 3 test cases are created to demonstrate how different indexes can affect the performance of query execution. For the simplicity of understanding, the measurement unit in these test cases is millisecond.

7. Recommendation

For already running Postgres database management system, DBA should do the following things to boost the database performance:

- check the database configuration files to see it is configured well
- use query logging utility to collect slow queries if Performance is degraded
- add necessary index to replace larger table sequence scan with index scan in the query execution
- use EXPLAIN for evaluating the use of indexes
- re-evaluated the application logic to reduce database access
- replace dynamic query with prepare query execution to enhance performance and security

Bibliography

Simon, R. & Hannu, K. 2010 PostgreSQL 9 Administration Cookbook
Published by Packt Publishing Ltd, ISBN: 978-1-849510-28-8 Quoted:23.01.2011

Lahdenmäki, T. & Leach, M. Relational Database Index Design and the optimizers
Published by John Wiley & Sons, Inc., ISBN: 13 978-0-471-71999-1 Quoted: 25.01.2011

Gregory, S. PostgreSQL 9.0 High Performance
Published by Packt Publishing Ltd, ISBN: 184951030X Quoted: 01.02.2011

DBTechNet 2010. DBTech EXT Index Design and Performance Labs URL: http://www.dbtechnet.org/labs/idp_lab/IDPLabs.pdf. Quoted: 08.03.2011

Dibyendu, M 2007. A Quick Survey of MultiVersion Concurrency Algorithms URL: http://simpledbm.googlecode.com/files/mvcc-survey-1.0.pdf Quoted: 02.03.2011

Enterprise DB 2008. Explaining Explain URL: http://wiki.postgresql.org/images/4/45/Explaining_EXPLAIN.pdf Quoted: 21.03.2011

Mullins, C. 2002. Database Administration
Published by Pearson Education Corporate Sales Division, ISBN: 0-201-74129-6 Quoted: 15.12.2011

Molina, H., Ullman, J. & Widom, J. Database System The Complete Book 2nd edition
Published by Pearson Education International, ISBN(13): 0978-0-13-135428-9 Quoted: 02.01.2011

Connolly, T. & Begg, C. Database System 5th edition
Published by Pearson Education, ISBN(13): 978-0-321-52306-8 Quoted: 12.01.2011

Matthew, N. & Stones, R. Beginning Databases with PostgreSQL 2nd edition
Published by Berkeley, CA: Apress. ISBN: 1-59059-478-9 Quoted: 03.02.2011

Ramakrishnan, R & Gehrke, J. Database management systems 3rd edition
Published by McGraw-Hill. ISBN: 0-07-246563-8 Quoted: 18.02.2011

Elmasri, R. & Navathe, S. Database System 6th edition
Published by Berkeley, CA: Apress. ISBN: 0-13-214498-0 Quoted: 16.12.2011

Wikipedia 2010a, PostgreSQL
http://en.wikipedia.org/wiki/PostgreSQL. Quoted: 05.12.2010

Wikipedia 2010b, Database tuning
http://en.wikipedia.org/wiki/Database_tuning Quoted: 15.12.2010

Wikipedia 2010c, phpPgAdmin
http://en.wikipedia.org/wiki/PhpPgAdmin Quoted: 11.01.2011

Wikipedia 2011b, Stored procedure
http://en.wikipedia.org/wiki/Stored_procedure Quoted: 13.01.2011

Wikipedia 2011c, vmstat
http://en.wikipedia.org/wiki/Vmstat Quoted: 31.03.2011

PostgreSQL 2010a, About
http://www.postgresql.org/about/. Quoted: 05.12.2010

PostgreSQL 2010b, Community Guide to PostgreSQL GUI Tools
http://wiki.postgresql.org/wiki/Community_Guide_to_PostgreSQL_GUI_Tools
Quoted: 14.12.2010

PostgreSQL 2010c, PostgreSQL 9.0.2 Documentation
http://www.postgresql.org/docs/9.0/static Quoted: 05.12.2010

PostgreSQL 2011a, Performance Optimization
http://wiki.postgresql.org/wiki/Performance_Optimization Quoted: 13.01.2011

PostgreSQL 2011b, GIN Indexes
http://developer.postgresql.org/pgdocs/postgres/gin-intro.html Quoted: 15.01.2011

PostgreSQL 2011c, How does CLUSTER ON improve index performance
http://www.postgresonline.com/journal/archives/10-How-does-CLUSTER-ON-improve-index-performance.html Quoted: 21.02.2011

PostgreSQL 2011d, A Generalized Search Tree for Secondary Storage
http://gist.cs.berkeley.edu/gist1.html Quoted: 21.02.2011

PostgreSQL 2011e, Tuning Your PostgreSQL Server
http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server Quoted: 04.04.2011

Microsoft 2010a, Optimizing Database Performance
http://msdn.microsoft.com/en-us/library/aa273605%28v=SQL.80%29.aspx Quoted: 15.12.2010

IBM 2010a, Database performance optimization tasks
http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=%2Frzatc%2Frzatcperformanceoptimatrix.htm. Quoted: 15.12.2010

Embarcadero 2010a, Database Performance Optimization
http://www.embarcadero.com/solutions/database-performance-optimization. Quoted: 15.12.2010

Frank, W. Performance Tuning PostgreSQL
http://www.revsys.com/writings/postgresql-performance.html. Quoted: 15.12.2010

stackoverflow. How Indices Cope with MVCC
http://stackoverflow.com/questions/4841692/how-indices-cope-with-mvcc Quoted: 15.01.2011

PHP Manual. Prepared statements and stored procedures
http://php.net/manual/en/pdo.prepared-statements.php Quoted: 14.03.2011

Linux User's Manual. TOP
http://unixhelp.ed.ac.uk/CGI/man-cgi?top+1 Quoted: 31.03.2011

The Linux Information Project. Context Switch Definition
http://www.linfo.org/context_switch.html Quoted: 02.04.2011

Mac OS X: How to View Memory Usage With the "top" Utility Quoted: 21.03.2011

pgAdmin 2011. pgAdmin 1.12 online documentation
http://www.pgadmin.org/docs/1.12/index.html Quoted: 05.04.2011

Attachments

Prepared Query Execution with PHP

```
<?php
/**
 * This php execution file is developed for
 * demonstrating how to utilze prepared query execution
 * */
$db = new db();

$query = "SELECT rowid, name FROM CUSTOMER WHERE status = $1 ORDER BY name LIMIT 10";

$db->f_prepare_query('list_customer', $query);
$res = $db->f_prepare_execute('list_customer', array('ex'));

while($row = $db->f_get_object($res)){
        echo "<br/> ";
        echo $row->rowid." | ".$row->name;
}
$db->f_free_resource($res);

class db{
        static $conn = NULL;
        public function __construct(){
                $conf = array(
                        'host'      => 'localhost',
                        'name'      => 'thesis_testing',
                        'port'      => 5432,
                        'user'      => 'username',
                        'pass'      => 'password',
                       );
```

```
                if(self::$conn === null){
                        self::$conn = pg_connect("host={$conf['host']}
                        port={$conf['port']} dbname={$conf['name']}
                        user={$conf['user']}
                        password={$conf['pass']}");

                        if(!self::$conn){
                        throw new Exception ('Can not connect to
                        database: '.$conf['name']);
                        }
                }
}

public function f_query($query){
                if(!$rnt = pg_query(self::$conn, $query)){
                        throw new Exception('Query is not valid:
                        '.$query);
                }
                return $rnt;
}

public function f_free_resource($res){
                pg_free_result($res);
}

public function f_prepare_query($name, $query){
                $name = pg_escape_string($name);

                $sql  = "SELECT name FROM pg_prepared_statements
                WHERE name = '{$name}'";
```

```
                    if(!pg_num_rows($this->f_query($sql))){
                        if(!pg_prepare(self::$conn, $name, $query)){
                            throw new Exception('Cannot
                            prepare query {$name} :'.$query);
                        }
                    }
                }

                public function f_prepare_execute($name, $parameters){
                    $rnt = pg_execute(self::$conn, $name, $parameters);
                    return $rnt;
                }

                public function f_get_object($res){
                    $rnt = pg_fetch_object($res);
                    return $rnt;
                }

                public function f_get_array($rnt){
                    $rnt = pg_fetch_array($res);
                    return $rnt;
                }
}
?>
```