Multi threaded query accelerator: A case study

Durgaprasad S. Pawar
Starent Networks Ltd.

Abstract:

As the amount of data in the database grows more and more it becomes necessary in most of the applications using database to organize it in a way which is optimized for both high data insertion rate and faster query execution. This paper presents a case study which describes a database design which uses data partitioning to store huge data and a lightweight multi-threaded application which works as a middleware for faster query execution (up to 13 times). More emphasis is given on the multi-threaded application which executes queries spanning across multiple tables, faster than only one single-threaded Postgres process.

Approximately 90GB of data is going to be supported by this implementation with queries spanning across entire data.

Data size and H/W specification:

~90 GB of data spread across more than 20 tables is stored in the database. The tables are partitioned on day basis and inheritance is used to enable easy maintenance of data. Each table contains ~20,000,000 to ~40,000,000 records.

The H/W used to run the application is 2 processor machine with 2 hard disks, 2GB of RAM 4GB of swap space (Sun-Fire-V245).

Need for speed:

The data is continuously flowing into the machine at a very high rate and is available on the machine's local hard disk. This data is then processed and entered into the DB in appropriate tables. Following precautions are taken during data insertion:

1. No indexes/constraints are present on the table
2. Bulk copy operations are used instead of single inserts.

Above measures have helped improve database insertion rate by more than 2 times.

Along with high data insertion rates, minimum query time is expected when queries are fired on the DB. The results of the query are dumped into a file and the file is then available as a report for data analysis.

Database can do it all? No:

High data insertion rates are managed by Postgres with its default configurations. However, we face following problems/limitations while using Postgres for querying:

1. Time taken to execute the application specific queries is very high since queries span across multiple tables with huge size and involve 'GROUP BY' and 'aggregation' operations.
2. For data spanning across more than 3 tables, Postgres throws 'out of memory' error when the query executing Postgres process size goes beyond 4GB[1].

---

[1]Since we are using 32 bit processor, 4GB is the max address space for a process

3. For querying on large data sets significantly more 'shared_memory'[2] and 'work_memory'[2] is required.

Multi threaded application functioning:

A multi threaded application is therefore designed to get results from multiple tables using different DB connections simultaneously and dump the result set periodically into a file. As an application developer we have complete control over 'out of memory' error because we can periodically dump the partially calculated result set into a file and free up the memory.

Following diagrams give a high level design of the application.

The main thread gets a set of usernames from all tables by querying their parent table. This list of usernames is in sorted order. Using 'constraint exclusion' the query is executed on the tables of required dates only [1]. This set of usernames is shared amongst all worker threads for further processing.

Each worker thread is associated with one table in the database which holds one day's worth of data. Each thread queries on its corresponding table and extracts the required data in sorted order of username. It uses cursors [2] for this purpose. Then this worker thread compares username in the received result set with the username in the list earlier prepared by the main thread. This is shown in figure 2.

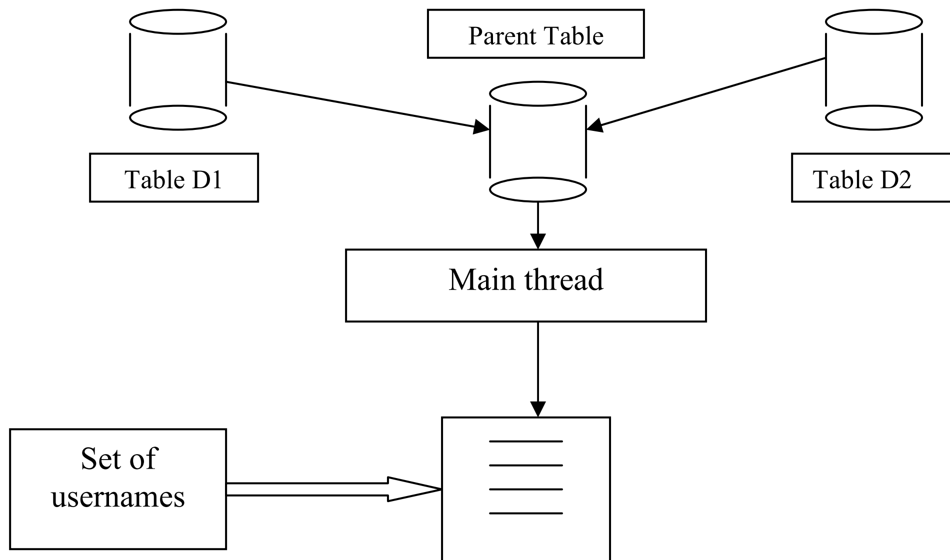

Fig. 1

Main thread also prepares a vector to store final result set. This vector is shared by all worker threads. Each thread writes the result set returned

[2] 'shared_memory' and 'work_memory' are configuration parameters in Postgres configuration file

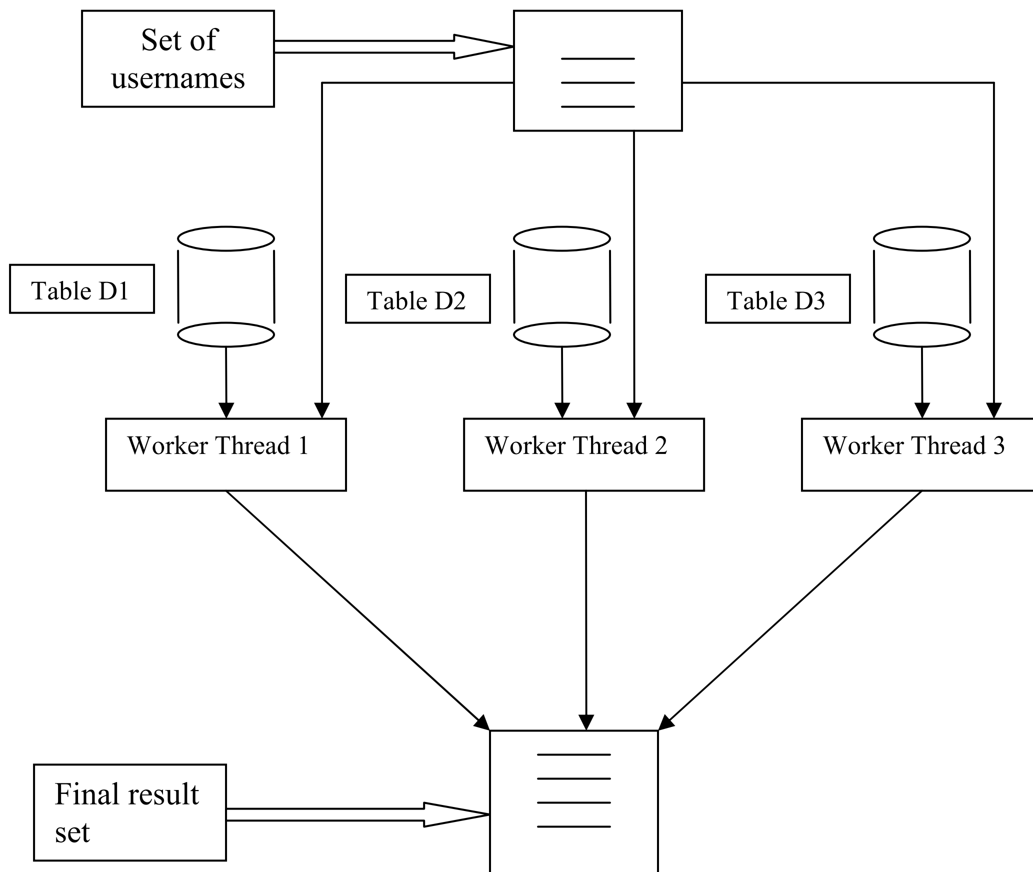

Fig. 2

by Postgres into this vector one row at a time. While writing, it also checks if a record for the same user already exists. If yes, then it updates this record instead of adding a new record. This is equivalent to the group by operation. Since the records are sorted by username already, the username lookup cost is almost negligible.

Another thread keeps a watch on this 'final result set vector' and as soon as it finds a row in the vector which is processed by all worker threads, it dumps it into a file. This frees up memory for more rows of the result set in the memory and prevents the application from going out of memory.

Query used by each individual worker thread is 'SELECT * FROM <tablename> ORDER BY <column name>'. This means, each thread makes only one sequential scan over all the records in the table. This reduces huge amount of memory required by Postgres for calculating large result sets. Because, now it has to simply return records from the table. Even with default parameters of Postgres, the result set is returned without going out of memory.

The gain in time for getting the final result set it astounding. It takes ~13 times less amount of time to get the result set into the file with this approach. 'Performance comparison section' covers the statistics.

We have developed this application as a middleware which will

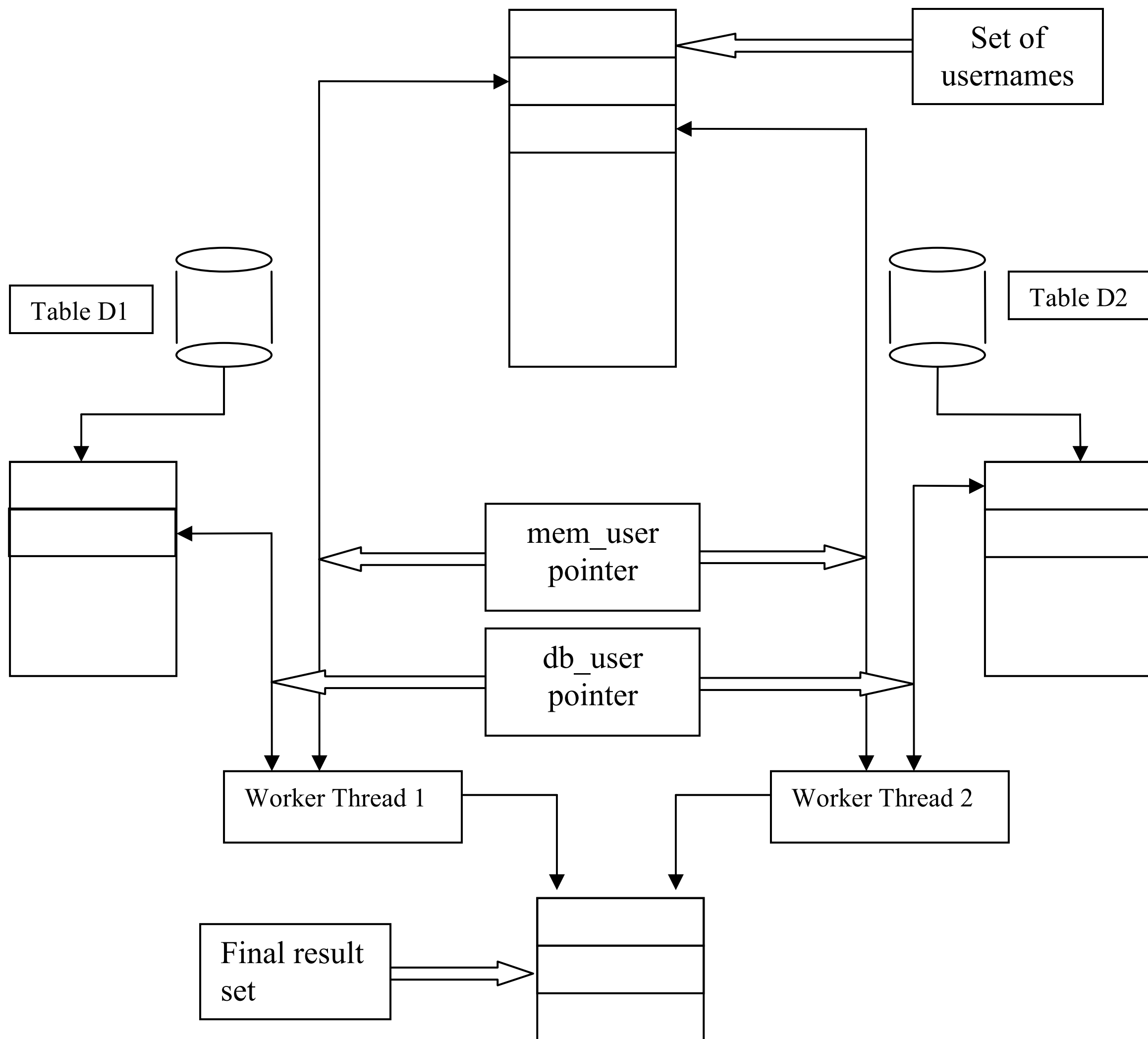

Fig. 3

get queries from other report generation application, parse it and execute it on the database and return the result set.

GROUP BY logic explained:

A simple algorithm for a worker thread will look as follows referring to above Fig. 3.

1. Point db_user pointer to the first username in the list of usernames extracted from the database table associated with this thread.
2. Point mem_user pointer to the first username in the list of usernames extracted by main thread.
3. Read Value(mem_user).
4. Read Value(db_user).
5. Compare db_user and mem_user and take actions based on the result as follows.
    a. Value(db_user) == Value(mem_user).
        i. Update/create this record in final result set and go to step 3.
    b. Value(db_user) < Value(mem_user)

i. Increment db_user and read Value(db_user).
ii. Go to step 5.

c. Value(db_user) > Value(mem_user)
   i. Increment mem_user and read Value(mem_user).
   ii. Go to step 5.

How Postgres helped:

The task of extracting common result set from database by main thread is made easier by table inheritance [3]. Also each table has a constraint defined on it which allows the queries to choose required tables only, when the 'constraint_exclusion' [1] bit is set.

The data accessed by worker threads is required in sorted order. Hence, table clustering [4] helps a lot in improving the performance of these queries. Clustering stores the data in the table in the order of the index on which it is clustered and thus allows fast data access and less swapping in and out of main memory.

Performance comparison and statistics:

1. Without table clustering:

Time required-
Using only DB: ~1680 sec (~28 min)
Using multi threaded application with DB: **~130 sec (~2 min)**

Following graphs show disk usage and CPU usage patterns of Postgres and Multi threaded application when the tables were not clustered on the index.

*(X-axis indicates time with each interval=2 sec,
Y-axis indicates Values)*

- Using only DB:

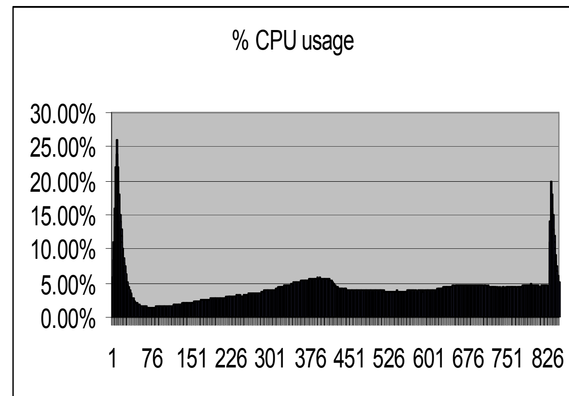

Fig. 4

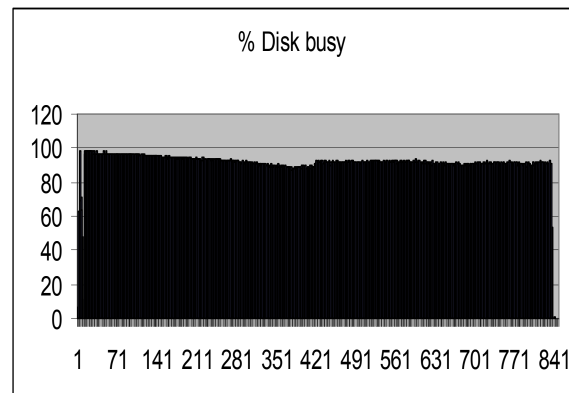

Fig. 5

- Using multi threaded application with DB:

Main thread-

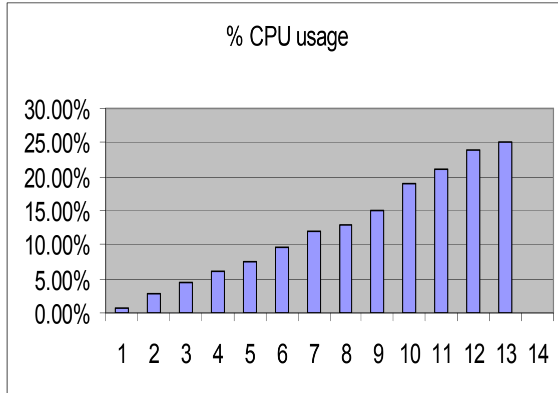

Fig. 6

Worker thread-

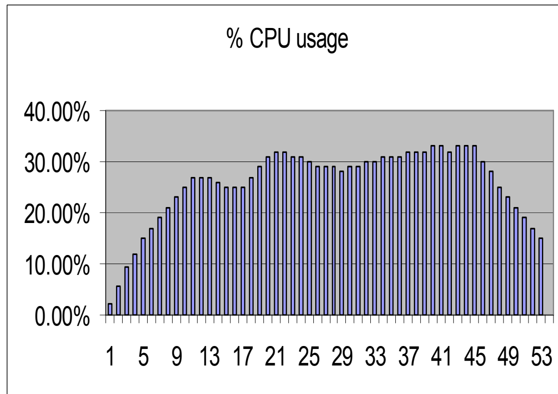

Fig. 7

Combined disk utilization of both types of threads-

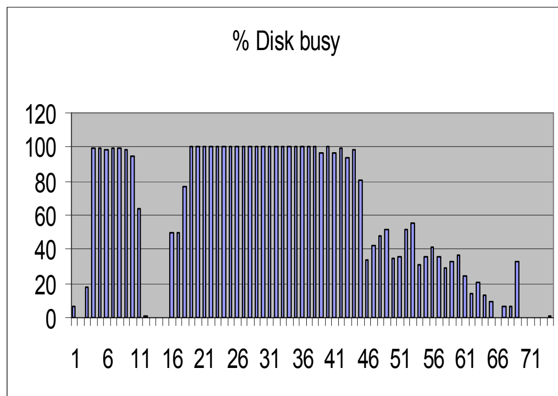

Fig. 8

The initial peak corresponds to extraction of common values for worker threads. The remaining part corresponds to worker thread's operation.

2. With table clustering:

The tables were clustered on the index.

Time required-
Using only DB: 1162 sec (~19 min)
Using multi threaded application with DB: **85 sec (~1.5 min)**

All the graphs have similar pattern as the earlier graphs. A combined disk utilization graph for both types of threads is depicted below.

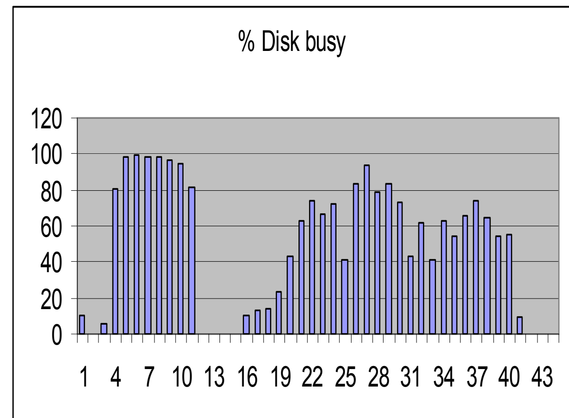

Fig. 9

The initial peak corresponds to extraction of common values for worker threads. The remaining part corresponds to worker thread's operation.

Disk read/write comparisons:

We also used Dtrace [4] scripts to see the disk read/write patterns when the table is clustered on an index and when it is not clustered.

The data used for this comparison was different from the data used for earlier tests.

1. Multithreaded application (with cluster):

As one can see from the graph, there is a smooth read operation going

on the disk. There are very less write operations indicating very less swapping.

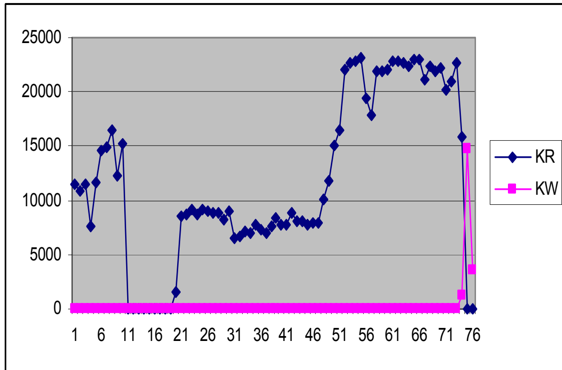

Fig. 10

Total data read: ~ 897 MB
Total data written: ~ 20 MB

2. Multithreaded application (without cluster):

In this case, the tables are not clustered on the index (username). As seen in the above graph, there is a lot of write activity going on along with the read operations. This indicates that there is a lot of swapping going on as the records needed are not available sequentially on the disk as in case of clustering.

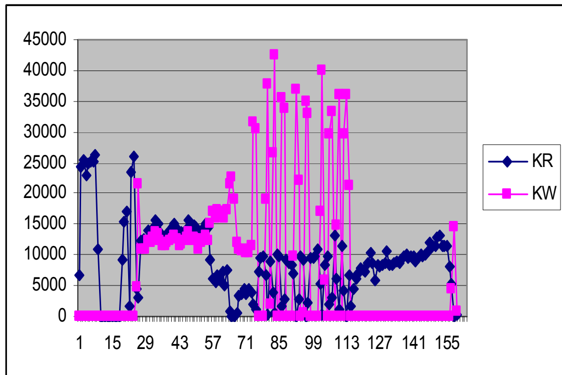

Fig. 11

Total data read: ~1214 MB
Total data written: ~1334 MB

Note that in the first case, the total amount of data written to the disk is approximately equal to the size of the report file that is generated in the end. On the other hand in the second case there was ~1334 MB data written to the disk as a result of swapping.

Advantages of Multithreaded application:

Following are the advantages of above mentioned approach over simply using Postgres for complex queries requiring aggregation operations big and complex queries.

1. Application developers control over 'out of memory' error.
2. Significant reduction in query execution time (up to 13 times), because single query broken into multiple simpler queries is executed by multiple threads on different tables simultaneously.
3. Less memory requirements (shared_memory[2] and work_memory[2]).
4. It keeps the disk less busy as compared to 'Use only Postgres' approach. Other I/O intensive tasks are benefited from this in our project.

Limitations:

1. The statistics and graphs above indicate that the CPU utilization is very high for this approach.

   The statistics were collected on quick implementation. With careful design of the application algorithm it can be brought down to lower levels.
2. For one day's data there is only one table and as per current design it will work at the speed of single Postgres process.

This can be overcome by letting multiple worker threads access different portions of the table and treat these portions as different tables.

Acknowledgement:

I thank Yateen Joshi and Pawan Shirbhate for their helpful suggestions and feedback. I also thank Starent for giving me time and providing necessary hardware for testing.

References:

[1] Table partitioning
http://www.postgresql.org/docs/8.2/interactive/ddl-partitioning.html
[2] Cursors
http://www.postgresql.org/docs/8.2/interactive/plpgsql-cursors.html
[3] Table inheritance
http://www.postgresql.org/docs/8.2/interactive/ddl-inherit.html
[4] Dtrace: How to guide
http://www.sun.com/software/solaris/howtoguides/dtracehowto.jsp