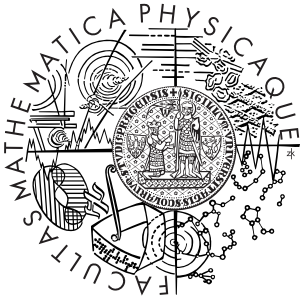
Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS



Bc. Daniel Vojtek

Implementation of parallel query processing in PostgreSQL

Department of Software Engineering

Supervisor: Mgr. Július Štroffek
Study programme: Computer Science, Software Systems

2010

I would like to thank my supervisor Mgr. Július Štroffek for his leading of my diploma thesis, for inspiring discussions and ideas. I would also like to thank Mr. Zdeněk Kotala for his active support in my initial attempts with PostgreSQL core system and for sharing of his deep knowledge in this field.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

I declare that I wrote my diploma thesis independently and exlusively with the use of the cited sources. I agree with lending the thesis.

Prague, 9th December 2010 Daniel Vojtek

Contents

Název práce: Implementace paralelního zpracování dotazů v databázovém systému PostgreSQL
Autor: Bc. Daniel Vojtek
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Július Štroffek
e-mail vedoucího: julo@stroffek.cz

Abstrakt: Paralelní zpracování dotazů je jednou z možností jak v reálném čase zpracovávat rychle narůstající množství dat uložených v databázích. Cílem této diplomové práce bylo prozkoumání možností, návrh řešení a zároveň implementace paralelního zpracování dotazů v open source databázovém systému PostgreSQL. Využil jsem návrhový vzor Master/Worker, kde standardní databázový proces vykonává pozici vedoucího. Jako pracovníky jsem využil procesy vznikající z řídícího procesu databáze, postmastra. V práci jsem se zaměřil na přípravu infrastruktury potřebné pro paralelní zpracovaní. Definoval jsem nový kontext nad sdílenou pamětí, který umožňuje efektivní správu alokací. Implementoval jsem vytváření pracovních procesů na základě požadavků vedoucích procesů. Definoval jsem také struktury umožňující jejich řízení. Následně jsem na této infrastruktuře implementoval paralelní operace, jako je řazení a SQL operátor UNION ALL. Výsledkem této diplomové práce je kromě implementace infrastruktury a několika paralelních operací také pojmenování problémů spojených s implementací paralelního zpracovaní, návrh jejich možného řešení a určení dalších možností rozvoje této implementace.

Klíčová slova: Paralelní zpracování dotazů, PostgreSQL, Master Worker paradigma

Title: Implementation of parallel query processing in PostgreSQL
Author: Bc. Daniel Vojtek
Department: Department of Software Engineering
Supervisor: Mgr. Július Štroffek
Supervisor's e-mail address: julo@stroffek.cz

Abstract: Parallel query processing can help with processing of huge amounts of data stored in database systems. The aim of this diploma thesis was to explore the possibilities, analyze, design and finally implement parallel query processing in open source database system PostgreSQL. I used a Master/Worker design pattern, in which standard PostgreSQL backend process is a master. As workers I used processes created from

postmaster. In the thesis I focused on preparing an infrastructure necessary for parallel processing. I defined a new top level memory context over shared memory, which allows efficient and convenient memory allocations. Then I implemented creation of new worker processes, based on master process requirements. To be able to control these workers I defined controlling structures using state machines. Then I implemented parallel sort operation and SQL operator UNION ALL using this infrastructure. The result of this diploma thesis is not only implementation of infrastructure and some parallel operations, but also description of the problems encountered during the implementation with their possible solutions. I also outlined further extensions of our implementation.

Keywords: Parallel query processing, PostgreSQL, Master Worker paradigm

Preface

The growth of databases has been exponential in recent years and considering the continuous fall in disk capacity prices, it is easy to assume that this trend will continue. However, we have to realize that traditional sequential processing paradigm is no longer sufficient. Consider that even with a data rate of 1 gigabyte per second it would take roughly more than 10 days to go through a petabyte database. Parallel processing can help with this enormous volume of data. First time used few decades ago and nowadays the parallel processing is provided by many commercial relational database management systems (RDBMS). This attention is a proof how important parallelism is for the future of processing huge volumes of data. In addition to traditional transactional data systems we can also see increase in deployment of decision support systems, data warehouses, data mining systems and online analytical processing (OLAP) applications.

Our intention in this diploma thesis will be to look at the actual state of open source RDBMS PostgreSQL, to identify possible uses of parallel query processing and implement some of them. Our implementation is probably not going to be anywhere close to being comparable with parallel execution support of other competitors, such as Oracle or Microsoft SQL Server. Our aim will be more at researching and developing running proof of concept that query parallelism is possible in this open source project.

In the first chapter we will overview parallelism in relational database systems and take a closer look at some commercial systems. In the second chapter we will focus on a PostgreSQL and provide some basic infrastructure necessary for any parallel processing to take place. In the third chapter we will show how parallel tasks can be implemented using this infrastructure. Finally, we will summarize and analyze results obtained from our implementation. We will also provide ideas about future development.

Chapter 1

Parallel Query Processing

Parallel databases are defined as database systems that are implemented on platforms which support parallel execution. In this first chapter we will give a brief introduction to parallel databases, parallel query processing and its motivations and objectives. We will also mention obstacles that are faced by developers trying to implement and use parallelism in database systems.

1.1 Motivations and Objectives

There are two basic motivations why the parallel processing is considered by database system developers. Main is the huge potential performance gains in some certain scenarios. The second is the effective usage of new available hardware on the market, which is very suitable to running tasks in parallel. As the hardware manufactures are now more than ever facing limitations of current resources and technologies used to build microchips and other components, such as transmission speeds or the density of transistors within a processor, new approaches had to be found. That is why the designers had to come up with a new idea on how to increase performance. They came up with a parallel processing. It is basically taking a large task and dividing it into multiple smaller pieces, all of them being executed simultaneously but each on a different processor. This should result in less time needed for an execution of that task. And also the parallel processing fits well on database systems as queries are run on a database. If we can split operations of that query to run on different segments/partitions of this database, then we have ultimately created parallelism. We can define these two goals for parallelism in database systems as the most important ones: Running queries on very

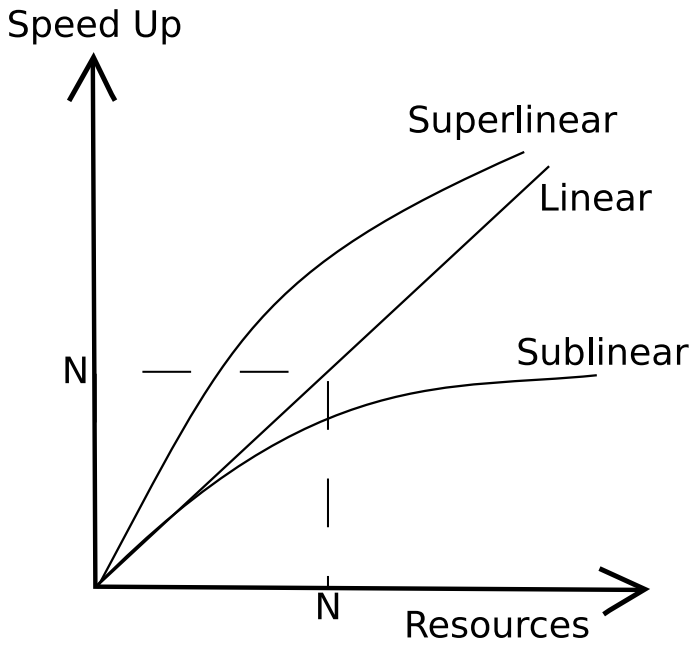


Figure 1.1: Speed Up

large databases and processing huge numbers of transactions per specific amount of time.

There are two measures in performance improvement. The first one is a throughput, which measures the number of tasks that database system can complete in a certain amount of time. And the second one is the response time, which as the name suggests, is the amount of time necessary to get a response/results after the request/query has been submitted. Considering a system which has to process a large quantity of small transactions, then to increase the throughput, it can process multiple independent transactions in parallel. But if the system has to process relatively small amount of very large queries, then it can improve the response time by performing subtasks of these transactions in parallel. To quantify these measures we define speed up and scale up.

Speed up refers to how much is parallel algorithm faster than a corresponding sequential algorithm. It is defined as

$$S_p = \frac{T_1}{T_p} \tag{1.1}$$

where:

- p is the number of processors,

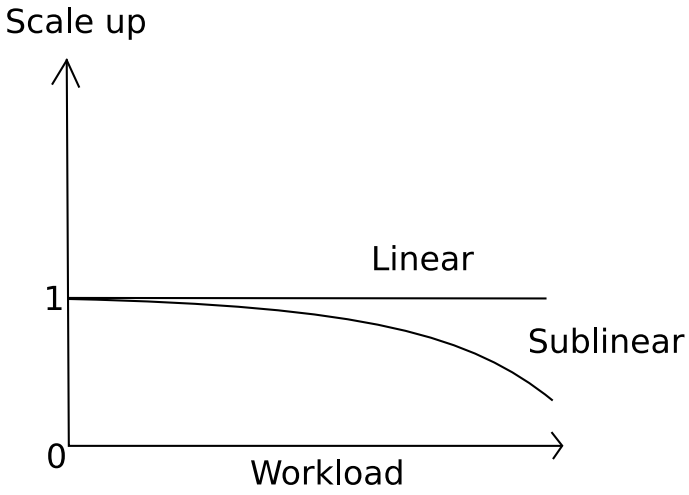


Figure 1.2: Scale Up

- T_1 is the execution time of a sequential algorithm,
- T_p is the execution time of a parallel algorithm.

Linear speed up is obtained when $S_p = p$, and it would mean that doubling the number of execution units will also double the speed. This kind of speed up is ideal, but more frequent is a sub-linear speed up. It is also possible to achieve a super linear speedup. For example when we add processors and also main memory can result in completely in-memory processing of the task. In the Figure 1.1 is shown comparison of different types of speed ups.

In the scale up we are measuring whether by adding extra resources we can process equivalently larger tasks in the same period of time. For example if our database table grew to 4 times of its previous size then if by adding 4 times more resources to the system we would achieve the same query processing time. The formula of scale up is:

$$S = \frac{T_u}{T_m} \tag{1.2}$$

where:

- T_u is the execution time of a smaller task on a uniprocessor system,
- T_m is the execution time of a larger task on a multiprocessor system.

Scale up of value 1 is called linear scale up. In the Figure 1.2 we can see a graph showing linear and sub-linear scale up. There are two types

of scale ups in parallel databases, based on how the size of the task is measured. Transaction scale up and data scale up. In transaction scale up the number of processed transactions is measured. For example when a single processor system was able to serve 1000 users running 10000 transactions per hour and now it has to serve four times more transactions. Then if after adding three additional processors, raising the number of processors to four, the time needed to execute all 40000 transactions would still be one hour, then we have achieved scale up of value 1 which is a linear scale up. However, if it would take 2 hours, then the scale up is sub-linear of value 0.5. Data scale up refers to the increase in size of the database in which the task is being run. This task has to depend on that size.

There are is a downside of using parallelism in the database systems. We will mention some of them. Such as additional cost when starting and finishing a parallel execution and skew. We have to initialize processes and other resources used in parallel execution during the startup. This could take longer than to execute some simple queries. At the end of the execution we might also need to consolidate all the results we have received from all units participating in the execution. While the task itself can run in parallel, the consolidation is usually performed by only one process, usually referred to as a master process. Both initialization and consolidation can not be parallelized so they can lower the overall speed up. This relates to the Amdahl law. It states that if P is the proportion of a program that can be made parallel and (1-P) is the proportion that cannot be parallelized, then the maximum speedup that can be achieved by using N processors is:

$$\frac{1}{(1-P)+\frac{P}{N}} \tag{1.3}$$

In the limit, as N tends to infinity, the maximum speedup tends to $\frac{1}{1-P}$. For example when 90% of the task can be run in parallel then the problem can speed up by a maximum of a factor of 10, no matter how large the value of N would be. So one important task is to reduce the part which can not be parallelized as much as possible .

Another problem is when processes share some resources and they have to compete for them. There is also a problem with skew, which is defined as an unevenness of workload partitioning. This happens when the data is unevenly distributed among processes, thus leaving some processes to run longer than others.

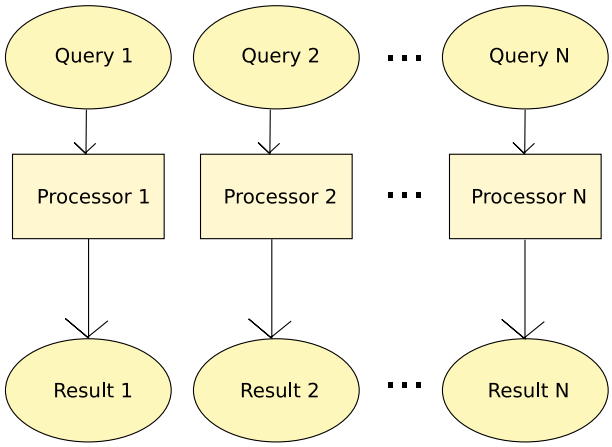


Figure 1.3: Interquery parallelism

1.2 Forms of parallelism

Parallelism for database processing can be implemented in many ways. Most common are:

- inter-query parallelism,
- intra-query parallelism,
- inter-operation parallelism,
- intra-operation parallelism.

The inter-query parallelism is what is called multi user access to database. Basically they are different queries in separate transactions running in parallel. It was primarily developed to provide a transaction scale up, i.e., to support large numbers of transactions per second.

In the Figure 1.3 we can see how each processor is processing its own query independently of other processors. Running this single-processor system will result in transactions/queries forming a queue, because only one can be run at any time. This would cause the execution time of a single query to be higher than of when it would have run alone. However, if we add more execution units, we can significantly improve overall transaction throughput. These transactions running in parallel might take longer to execute in comparison with them being run in isolation.

Intra-query parallelism refers to situation in which a single query is being run in parallel on multiple processors and disks. This is especially beneficial for very long running queries, where inter-query parallelism

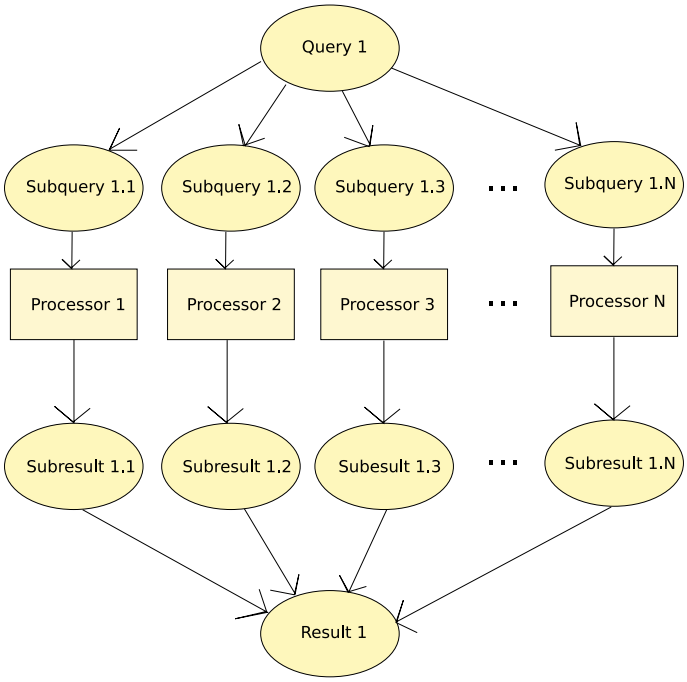


Figure 1.4: Intraquery parallelism

can not help to speed up the execution. The Figure 1.4 is an illustration of intra-query parallelism. There is a long query split into n sub-queries. Each sub-query is then processed on a different processor and produces its own results. These results usually have to be consolidated before they can be returned to the user. E.g., if sub-queries consist of sorts on randomly created data partitions then the final sort has to be performed. Intra-query parallelism can be done using intra-operation parallelism, inter-operation parallelism or combination of both.

In the intra-operation parallelism we parallelize individual operations of the query execution plan by running them on different subsets of data/table. This form of parallelism can also be referred to as a partitioned parallelism. With the large numbers of records in a table, the degree of parallelism can also be very high and that is what makes this form highly suitable for database systems. The Figure 1.5 is based on the previous figure and shows how the operation, which is part of a subquery, is processed on a different partition to create a parallelism. Hence it can be described as a Single Instruction Multiple Data, because the same operation is executed on different data parts. However, two serious issues emerge with this form. The first is how the operation can be arranged so it can execute on different data partitions and the second,

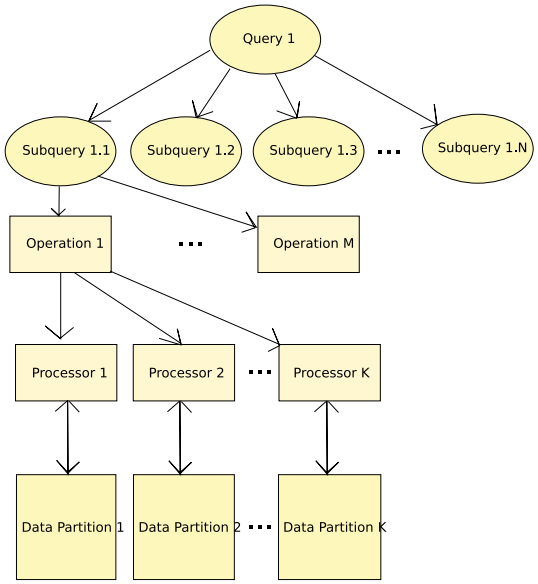


Figure 1.5: Intra-operation parallelism

how the data should be partitioned to be able to work on it. Because of this, the database system has to provide a new parallel implementations for basic query operations such as parallel search, parallel sort, parallel group by/aggregates or parallel join.

The next form is inter-operation parallelism in which we simultaneously execute multiple operations within one query/transaction. There are two basic forms: pipelined parallelism and independent parallelism. It can also be useful to use a combination of both. In the pipelining, as the name suggests, two or more operations are pipelined, i.e, output of one operation is consumed by a second operation, even before the first operation has produced a whole result set. This can be very useful when we want to skip writing intermediate results and forward them immediately to consuming operation. The Figure 1.6 shows how operations from 1 to k form a pipe. It might seem as sequential processing but those operations are all executed concurrently. Drawbacks of this form are that the length of a pipe is a decisive factor in the degree of parallelism. Also only operations that do not need to wait for complete result set can form a pipe. Operations like sort have to wait until all the records are present so that they can perform their task correctly. However, some parts of sorting can be done on the fly, e.g., in external sort the final merge phase. Another disadvantage of this form is when pipelined operations are not equally fast. If some operation takes considerably longer to execute then

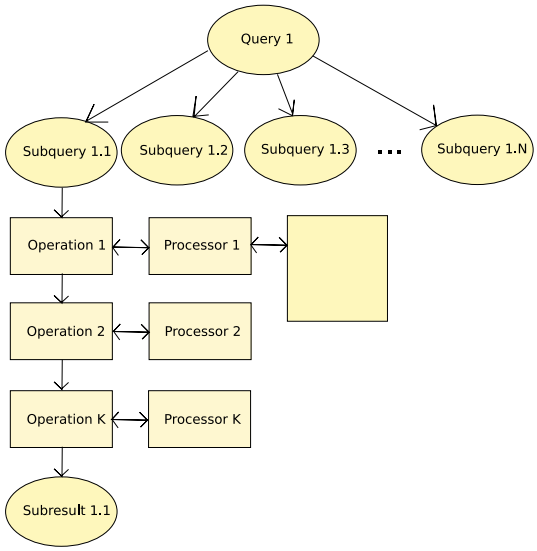


Figure 1.6: Pipeline parallelism

the others, it will lower the overall speed of processing records. And finally there is the independent parallelism. Consider a query with four tables joined together to form a result. In that case we can join Table 1 with Table 2 in parallel with join of Tables 3 and 4.

As all the forms have their positives and negatives, often a combination of both can be more suitable. As an example we will join four tables, namely Table 1 to Table 4. With a joining order 1 to 4 and joining attributes in all pairs of consecutive tables. One possible way to parallelize this query is:

- Independent parallelism. Run two joins in parallel - Table 1 with Table 2 and Table 3 with Table 4. These two operations will form Result1 and Result2.

- Pipelined parallelism: Records from both Result1 and Result2 are as soon as they are created sent to the next join operation forming a pipe.

- Intra-operation parallelism: All three mentioned joins are executed on partitioned data on multiple processors.

Scheme of this processing is in the Figure 1.7.

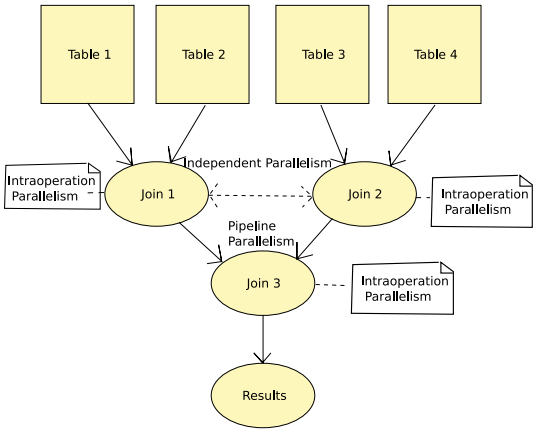


Figure 1.7: Mixed parallelism

1.3 Architectures

The need of a performance improvement is only a half of what drives the database parallelism. It is also heavily supported by recent changes in hardware with a wide spread of parallel computers. They are no longer domain of only supercomputers. They are now available in many forms, such as symmetric multiprocessing (SMP) machines, high speed multi core server systems, clusters of SMP machines, and massively parallel processors. In this chapter we will briefly review basic architecture styles as they have a significant impact on overall architecture of every database system. With each one having its positives and negatives, there is still no widespread consensus in which one is superior to all others, in terms of usability for parallel processing. There are four classes of parallelism based on what is shared among the processes:

- shared-memory,
- shared-disk,
- shared-nothing,
- shared-something.

Shared-memory architecture is an architecture in which all available processors share both main and secondary memory. Tasks are then spread into multiple processors. The number of processors is usually in some relation to the maximum degree of parallelism. Processor load balancing is simpler because all the data in main memory can be easily

shared and processors can ask for it as they wish. However there might be a problem with a bus contention as all processors are competing for this resource. This means that the number of execution units is somehow limited to a point that the bus can handle.

In a shared-disk parallel machine all processors have an access to the same disks but can not access each other's local main memory. One key advantage of shared-disk systems over shared-nothing is in usability, since database administrators of shared-disk systems do not have to consider partitioning tables across machines. On the other hand, a very big problem is a congestion of the network because all processors are accessing the same disks. Another feature of the shared-disk architecture is that the failure of a single DBMS processing node in a cluster does not affect the other nodes' ability to access the full database. This is in contrast to both shared-memory systems that fail as a unit, and shared-nothing systems that lose at least some data upon a node failure. As there is no partitioning of the data in a shared disk system, data can be copied into main memory and modified on multiple machines. Unlike shared-memory systems there is no place to coordinate this sharing of the data as each machine has its own local memory for locks and buffer pool pages. Hence there is a need to explicitly coordinate data sharing across the machines.

Shared nothing architecture has no problem accessing the data as each processor has its own primary and secondary memory. However, load balancing is more complicated as different processors might have different load. Shared something architecture is a mixture of shared-memory and shared-nothing architectures.

Another important topic that has to be considered are interconnection networks that ranges from buses, meshes to hypercubes.

1.4 Other RDBMS

In this section we will take a closer look at parallelism support in Oracle Database Server and Microsoft SQL Server. As this is all proprietary software, the source code is not open. We will therefore focus only on main models and ideas.

1.4.1 Oracle

Oracle is an example of the shared everything architecture, while allowing to build a cluster of computers to run on this single database. Because

of shared everything architecture it is not necessary to have a predefined data partitioning. However, they can use it to optimize the performance if it is present. Operations that can be executed in parallel include:

- SQL based data loads.
- Queries.
- Recovery backups.
- Index building.
- Statistics gathering.

As we are interested in queries we will focus on them. SQL parallel execution in Oracle is based on master worker model. Master is in Oracle called Query Coordinator (QC) and workers are called parallel servers. Started session is actually the QC and parallel server are then the different sessions performing the work. QC performs only minimal work, such as when query has to compute a sum, then the QC will calculate the final value by adding values from parallel workers. But its main task is the distribution of work to parallel servers. Oracle provides pooling of parallel servers so it can reduce the initialization costs of parallelism. Another interesting concept they adopted, is a pair of slave sets (parallel servers). When the operation is parallelized then it requires two slave sets. One producing rows and the other one consuming them. Producing rows also requires redistribution algorithms. This pair of sets has a consequence in how the degree of parallelism (DOP) transforms to number of spawned parallel processes. So when the DOP is four then there are actually two sets of four parallel workers. Exception to this pair paradigm is only in simple queries like counting rows in one table which requires only one set.

When the operations access the data then the smallest unit is called granule. Oracle uses to distribute the data so-called block-based granules which are ranges of blocks on disk. Besides of these block-based granules there are also partition-based granules which take advantage of partitioned tables. But because of the rule that only one server performs the work for all the data in a single partition, the Oracle will use this type of granules only when the DOP is smaller then number of partitions accessed in operation.

As parallel operations require data redistribution, they have implemented various types of it. Most common ones are

- Hash - redistribution based on computing hash function over the data used for equal distribution.
- Broadcast - sending data to all parallel servers of consuming pair. Used when one of the result sets in join operation is much smaller then the other result set.
- Range - generally used in sorts as the data does not need to be rearranged.
- Key - ensures that the result sets for individual key values will be clumped together.
- Round robin - when no redistribution constraints are required then this can be used.

Parallel execution in Oracle can be allowed by hints in queries, setting attributes on tables or allowing it for specific session. Other useful features consists of load balancing and monitoring of parallel execution.

1.4.2 Microsoft SQL Server

Microsoft's approach in parallel execution is in two areas. Massive parallel processing (MPP) systems, for which they have a product called Parallel Data Warehouse, and standard parallel query processing. Parallel Data Warehouse implements hub and spoke architecture, in which the MPP system with a database is a hub and the spokes are data marts. The data mart is a subset of the data warehouse, usually oriented to a specific business line or a team. This is in opposition with traditional approaches, such as monolithic data warehouses or federation of independent distributed data marts.

The implementation of a parallel query processing on SMP machines is a feature in Microsoft SQL Server. They are aware that not everything is suitable for parallelization, so they are using a couple of system settings to determine whether parallelism can be used:

- Max Degree of Parallelism - this defines the number of processors the SQL Server can use when executing a parallel query.
- Cost Threshold for Parallelism - a minimum cost for operations that may run in parallel, this should prevent parallelism for fast queries.

As the optimizer assigns costs to the operations in a plan, this cost is then compared with a threshold and only when it is greater the operation may be defined as parallel. The actual decision process as described in [2] is as follows:

- Check the number of processors the machine has - must have more than one.
- Check number of available threads - operating system must have available threads.
- Check the type of a query or index operation that is being performed.
- Estimate the number of rows to process - number directly affects the cost.
- Check the statistics - optimizer may decide to not use parallelism or lower the degree of parallelism if the statistics are not up to date.

When all the conditions are met the optimizer adds marshalling and control operators, called exchange operators. They include Distribute stream, Repartition stream and Gather stream. After these operators are inserted the result is a parallel-query execution plan. For example consider a query returning sales by products:

```
SELECT  [so].[ProductID]
       ,COUNT(*) AS Order_Count
FROM    [Sales].[SalesOrderDetail] so
WHERE   [so].[ModifiedDate] >= '2003/02/01' AND
        [so].[ModifiedDate] < DATEADD(mm, 3, '2003/02/01')
GROUP BY [so].[ProductID]
ORDER BY [so].[ProductID]
```

In the Figure 1.8 we can see generated parallel plan for this query. It starts by reading rows using standard clustered index, but afterwards the exchange operator Repartition Stream takes these rows and outputs them as multiple data streams. There is also a more detailed XML description of every plan with exact numbers of used execution units. This operator has to repartition incoming data based on ProductId so that the results computed by worker threads are correct. Also if we realize that the results should be ordered then the Repartition Operator should split the data based on range of values. This resembles the already mentioned Oracle distribution types.

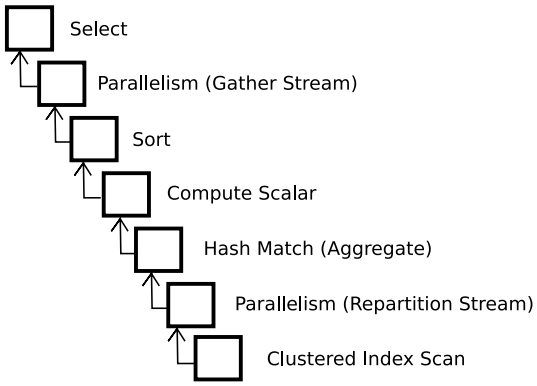


Figure 1.8: SQL Server Parallel Plan

Chapter 2

PostgreSQL

In this chapter we will give an overview PostgreSQL database system, its capabilities, architecture, already present parallel features and introduce our infrastructure for parallel execution in a context of this system. This will include dynamic usage of shared memory, work definitions, process of spawning new processes and state based communication between master and worker processes.

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2, developed at the University of California at Berkley Computer Science Department. It is an open-source descendant of this original Berkeley code. It supports a large part of the SQL standard and offers many modern features such as:

- complex queries,
- foreign keys,
- triggers,
- views,
- transactional integrity,
- multi-version concurrency control.

Also, PostgreSQL can be extended by the user in many ways, for example by adding new:

- data types,
- functions,
- operators,

- aggregate functions,
- index methods,
- procedural languages.

From a software developer point of view it is an excellent source of wisdom how mission critical, efficient and very stable systems should look like. Source code is written in C language with some Assembler code for further optimizations and it is very well documented. Every new feature or bug fix goes through rigorous discussion in a community forum and all source code changes are always reviewed by core developers as they only have a write access to main source code. So the system stays stable even after many years of intensive development. Because everything has to be reviewed by the community, it makes it also very difficult to make any larger changes to the source code. But no one can prevent you from developing your own distribution as many companies have done so. Even if your work is rejected by the community, as it is in most cases, everyone is free to develop its own changed version.

2.1 Architecture and Internals

PostgreSQL uses a process per user client/server model. Any PostgreSQL session consists of following cooperating processes:

- A supervisory daemon process called postmaster.
- Front end application the user used (psql, ...).
- Backend database server (postgres process).

The Postmaster manages multiple databases on a single host. This collection is called database cluster. A front end application which needs to access a given database of the cluster make calls to the library (libpq, jdbc, ..). The library connects over the network with the postmaster. Postmaster starts, by forking itself, new backend server process and connects it to the front end application. From now on, the front end process and the backend server communicate only with each other, without any intervention from the postmaster. With the only exception of sending a cancel query signal.

The Following stages need to happen when user wants to get the results of a query:

- A connection from an application program to the PostgreSQL server has to be established. The application program transmits a query to the server and waits to receive the results sent back by the server.
- The parser stage checks the query transmitted by the application program for correct syntax and creates a query tree.
- The rewrite system takes the query tree created by the parser stage and looks for any rules (stored in the system catalogs) to apply to the query tree. It performs the transformations given in the rule bodies.
- The planner/optimizer takes the (rewritten) query tree and creates a query plan that will be the input to the executor.
- The executor recursively steps through the plan tree and retrieves rows in the way represented by the plan. It makes use of the storage system while scanning relations, performs sorts and joins, evaluates qualifications and finally hands back the rows derived.

2.2 Parallel infrastructure

In this section we will cover all parallel infrastructure that we have written and also give a reasons for decisions we have made. It might seem that parallelism is a new feature to PostgreSQL source code but in a closer look we can see that it is not true. Overlooking the obvious inter query parallelism, hidden by multi user access, there are for example vacuum workers. These in fact execute single task of vacuuming a whole database in parallel. This task is divided among them and they concurrently executes it table by table. Postmaster plays the role of the master process which coordinates workers. Another parallelism which in fact might use threads is hidden within cryptographic library that PostgreSQL uses.

2.2.1 Threads vs Processes

The very first question we had to deal with was whether to use threads or processes or some combination of both. In the beginning we focused on threads. Their positives are shorter startup times and no special handling of shared data as all the data in a process is shared among all threads. However, the shorter startup time can be minimized in case we will have a pooling for workers. This is regardless of whether they will

be threads or processes. When we realize that parallelism make sense mostly in CPU intensive tasks which will take some nontrivial amount of time to execute, then it makes the startup time or generally any constant overhead connected with parallelism to be less significant in comparison with the execution time of the task itself. The other benefit of using threads, inherently shared all global data structures within one process, is also one of the biggest problems in software projects that were not intended to be thread safe. And because the PostgreSQL source code is generally not thread safe, it would require to review the code for all the possible risks, such as using global variables as caches. Taking into the account the scale of source code this is a nontrivial task to do. But it has to be considered because sharing large volumes of data between processes will be very costly in terms of execution time of query. The community has also raised objections against using threads. This has been declared in many discussions and main argument was that the threads are not standard on all the platforms that currently PostgreSQL supports. Even though this argument could be overcome by declaring that this functionality will be available on only a subset of platforms it is very probable that it will be as such refused to be added to the core source code. Since it cannot be implemented as a contrib module this will mean that our work has no chances of reaching the real distribution. Apart of the obvious problem of shared variables and possible race conditions there are also design problems. For example locking mechanism. Currently all the locks held in the system are associated with some process. Even when we are going to allow holding only read locks during the operations that run in parallel it means that we would have to re-implement it so that the locks are associated to threads and solve possible deadlocks inside of one process. This would not be a problem if have used processes instead. And there are many more such designs that will not work with threads.

The performance improvement is the main goal of using parallelism and the possible speed reductions when transferring data between processes. We therefore started our implementation based on threads. However, we abandoned this approach as we received more negative comments from the community and as we faced more and more difficulties with no thread safe source code. We finally chose processes as the worker execution units in our implementation.

2.2.2 Shared memory

With the decision to use processes we had to implement an effective and easy to use mechanisms for sharing data between them. The operating

system provides some options for inter process communication, such as:

- shared memory,
- pipes,
- sockets,
- message passing,
- semaphores.

The most suitable for us is a shared memory as it is already heavily used in PostgreSQL system to store all the critical data. Shared memory is allocated as one region during the startup of postmaster. Size of this region is computed as a sum of requests from all the modules participating in the database system. There are also options how to increase this size inside contrib modules. The size of this region can not be changed on the fly, system has to be restarted. Because all the processes participating in database system are forked from postmaster, they all have access to this memory and all references are valid. Historically there has been an option of running a stand alone backend. It had to map this shared memory region and so all pointers had to be kept as offsets from starting address of mapped shared region. But this has changed during our work on the thesis and now all pointers to the shared memory are valid, making the offsets obsolete and deleted from source code.

To allocate something in this memory PostgreSQL provides some structures such as very simple queue or a hash table to which we can add our structure under specified string. Other processes are then able to find and access this structure under that specific string. This would be very cumbersome and ineffective to use. We would ideally need a malloc style access to shared memory. PostgreSQL has adopted a very nice design of memory contexts in which all allocations are made through palloc (postgre malloc) functions. We will describe the memory context in the next paragraph and show how we took advantage of this concept.

PostgreSQL allocates memory within memory contexts, which provide a convenient way of managing allocations made in many different places that need to live for a different amount of time. Destroying a context releases all the memory that was allocated in it. Thus, it is unnecessary to keep a track of individual objects in order to avoid memory leaks. Instead, only a relatively small number of contexts has to be managed. There are these top memory contexts:

- TopMemoryContext.
- ErrorContext.
- PostmasterContext.
- CacheMemoryContext.
- MessageContext.
- TopTransactionContext.
- CurTransactionContext.
- PortalContext.

Every new context will become a child context of one of these, or their sons. So the contexts form a standard tree structures. During the destruction of a context all its child contexts are also recursively destroyed. And all allocation made on them are released. This way no context will be forgotten and no memory leaks will occur. All of these memory contexts are based on a standard allocation via malloc and free functions. But they provide some optimizations such as instead of immediate freeing after calling pfree they just keep the memory inside list structure for future palloc calls. This way allocations and deallocations are faster. On the other side, it requires some additional data space appended to each requested memory. We have to know the size and the context in which this allocation was made so it can be correctly freed. So each pointer returned in palloc call is then a little bit larger.

We have implemented a combination of both of these approaches. A new top memory context which allocates requested memory in a shared memory. This memory is allocated as a single region and all subsequent callings of palloc take part of this memory. Another advantage of PostgreSQL context API is hiding implementation details from palloc caller. Instead of writing our own memory manager we were able to easily reuse public library OSSP mm. As its documentation states, it is a 2-layer abstraction library which simplifies the usage of shared memory. It hides all platform dependent implementations details. We basically don't need it because PostgreSQL already has a shared memory region allocation routines, but it will not cause us any trouble. However, the second level is exactly what we need. It provides a convenient high level malloc style API. This library basically keeps track of all allocations with its sizes and manages them in a way PostgreSQL manages allocations requested

by palloc calls. The result is a very efficient management of memory allocations. We only had to provide initialization of our new context which forwards creation of shared memory region to this library and also forwards palloc calls. However, to satisfy the memory context API we had to add to each pointer its allocating context and size. This is a prerequisite for routine pfree as it will need to know where it should return this pointer. The already mentioned possible child contexts will share allocated region by the top level context. This could be changed in the future if necessary. Also when we run out of shared memory it will raise an error but it could easily be re-implemented to allocate additional region if necessary. This would of course had to be configurable by the user as he should have options to control the memory usage.

On top of this new memory context we have also build structure for transferring data between master and worker processes, generally between any two processes, which we call shared buffer queue. This structure is slightly modified implementation of classic producer/consumer scenario with finite buffer size. PostgreSQL already has a queue in shared memory but this lacks any kind of concurrent access protection, basically they are just linked pointers. The maximum size of the buffer is a configuration parameter. Users are therefore free to experiment with different settings. As a synchronization primitive we have used semaphores. They are standard means for synchronization and they provide passive wait. This is very useful when we want the producing process to sleep when the queue is full and as well in case of empty queue for the consumer. However, in case we have one consumer and multiple producers and we don't care from which producer we take next data, we could specify this in argument in get routine and if the queue is empty the call will return immediately with false return value. This way we can go and check other producers if they have something ready for our consumer. The choice of synchronization primitive was difficult as PostgreSQL is very strict in using semaphores. In fact, each backend process uses only one semaphore to sleep on during the wait for lock. This is mostly historic issue when the number of semaphores was very low in operating system. Instead whenever the access has to be protected from concurrent modification the programmer has to use structure called spinlock. These spinlocks are defined in Assembler for each platform Their main purpose is to provide a very fast synchronization primitive which will not need to call any core routine. This can work only if we can block the signal handling, but the documentation states that blocking is not necessary when we intend to hold a spinlock for very short time. With our decision to use standard

semaphores we had to increase the internal limits on the count of initialized semaphores as their total number is strictly checked with each new semaphore.

2.2.3 Workers and Tasks

Important part of infrastructure is "on demand" creation of workers requested by master backend processes. These worker processes will have to do the job currently done by master backends. It is therefore only natural that their creation together with all the initialization steps should be very similar to the creation of standard backend processes. That is why master process can not just fork himself but has to ask postmaster for new processes. Master backend therefore writes his requests to special shared structure in the top shared memory context and sends signal to the postmaster. These requests contain:

- Task type.
- Flag indicating whether the task was processed.
- Structure with parameters.
- Structure for results.
- Structure of worker (include its state).
- Unique identification of the task.

As we have already said, the initialization process of master and worker backend process is very similar. It differs for example in connecting to database or authentication. Other parameters include one shared buffer queue for sharing data and also one structure specific to the task being performed. For example in sorting, we would need to send the parameters of sorting, such as on which columns it should be sorted, the direction of sort, or whether the nulls come first. The newly created process then communicates with a master by writing to Worker structure, primarily by changing its state. Access to this structure must therefore be protected, in this case we used a spinlock. The whole process is shown in the Figure 2.1.

One of the things that is associated with every worker is its state. It defines its life cycle and is also used for synchronization with master backend process. When master backend process asks postmaster for some

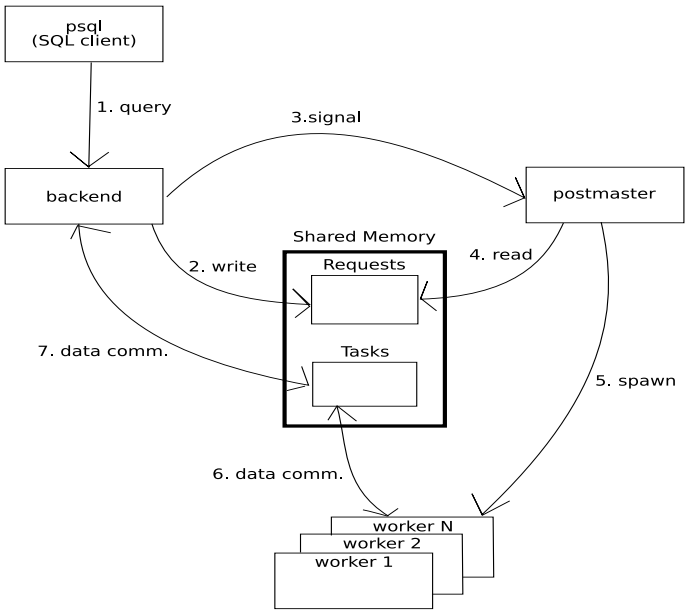


Figure 2.1: Workers creation

new workers, they will be created in an initial state. Master then periodically checks for workers in this specific initial state. Another important case when synchronization is necessary, is in ending workers. We have to be able to correctly release all shared resources, such as the shared buffer queue. This can be easily achieved by defining a state and associating it with a situation that workers are no longer using any of these resources. Then we can freely dispose of those resources. State machines can also be different for each type of task.

Apart from the shared buffer queue for sending tuples we might also need to allocate some other structures in shared memory. This memory will then have to be released after the workers have finished their tasks. To ensure that we will not have to release all the pointers one by one, we have defined a new a child memory context from top shared memory context for each master backend process. All allocations in shared memory, except the tuples transferred in shared memory queue, are then done in this context. This is because workers do not know of memory contexts created in master as all workers are forked from postmaster. As a consequence worker backends can not also release this memory. For example task parameters will be allocated in this child context. It will become very important in a situation when we will want to transfer a part of execution plan as it can be a quite complex structure. But releasing this memory will be done easily by destroying this child shared

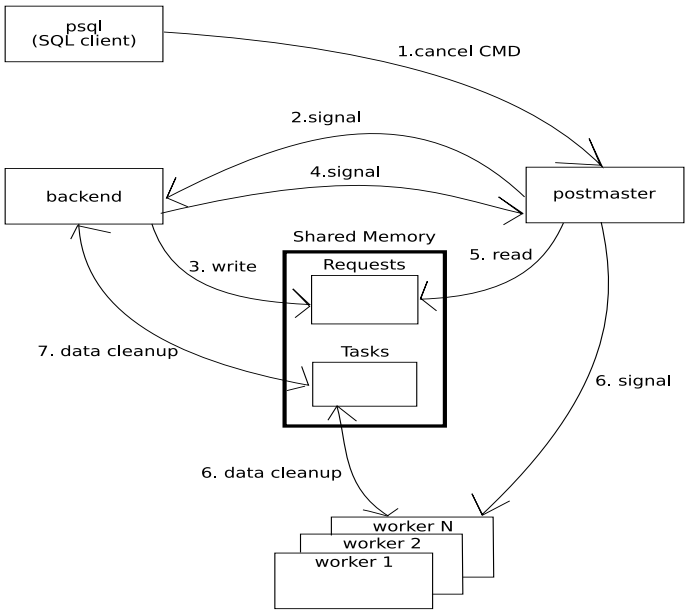


Figure 2.2: Cancellation

memory context.

It becomes a little bit more complicated when a user decides, that he does not want to run the query anymore. He declares it by sending a cancel signal to his front end application. We have to guarantee that all the workers will end their execution and correctly release all shared memory they used. Here we will also take advantage of a rule that workers allocated in shared memory only tuples send via shared buffer queue. So the master cleans all used queues and only release its shared memory context. This whole process is shown in the Figure 2.2. We can see that the master after receiving a cancel signal forwards it to the postmaster, which then sends it to workers.

With this we have prepared basic infrastructure using which we will implement parallel operations. In the next chapter we will show some example implementations using this infrastructure.

Chapter 3

Implementation

In this chapter we will present implementations using our infrastructure described in the previous chapter. As an example of intra node parallelism we will show parallel sort. Inter node parallelism will be represented by parallelized append node which executes UNION ALL part of SQL query.

3.1 Sorting

Sorting is one of basic operations in the database processing used whenever ORDER BY or DISTINCT is present in a SQL query. All common used non parallel algorithms are very well documented. They belong to two categories, internal and external sorting. The distinction is based on whether the sorting takes place entirely in main memory of a computer or not. Quick-sort is example of internal sorting and external merge sort is example of external sorting. These algorithms are important also in parallel processing because they are used in a local sort on a node participating in parallel execution or they can be easily modified to support parallel execution. From the database processing point of view, where large amounts of data are common, the more important ones are parallel external sorting algorithms. These include:

- Parallel Merge-All Sort.
- Parallel Binary-Merge sort.
- Parallel Redistribution Binary-Merge Sort.
- Parallel Redistribution Merge-All Sort.

- Parallel Partitioned Sort.

Parallel merge-all sort as described in [1] is composed of two phases. In the first phase called local sort phase a local sorting is carried away in each processor. Usually by a serial external sorting algorithm. After this phase, the final merge phase starts, which uses a standard k-way merge algorithm. However as it is carried out by one processor it imposes an obvious problem of heavy load on it during this phase. Also network contention is a problem because all locally processed data have to be transfered to the one processor in final phase. Parallel Binary-Merge sort tries to overcome this problem by pipelining the final merge phase. This means that the results from a pair of processors are merged by one of them and so one until there is just one pair. That is where its name Binary comes from. It is better, but there is still a problem at the end when there is just one processor to do the final merge. Also much more network connection is used as the data is transfered multiple times. There might be also limits on open files, but this algorithm can handle it much better than previously mentioned merge-all algorithm.

Parallel redistribution binary-merge sort is similar to parallel binary-merge but differs in the number of processors involved in final merge phase. Now all processors are used on all levels as the data is first re-distributed and then merged. For example after initial local sort each pair of processors redistributes data based on value range among themselves, do the merge, and redistribute to four processors and repeat until redistribution to all processors. Then each processor has a part of the result and no final merge is necessary. This algorithm has also problems with network overload and can as well suffer from data skew as some processors will have to merge much more data then others.

Parallel redistribution merge-all sort differs from standard parallel merge-all sort in that it after the initial local sort redistributes the data to all processors to do the merge. It may suffer from the same problem as the standard version, when there are limits on the number of open files.

In the parallel partitioned sort we first redistribute the data and then do the local sort. Main benefit is then that there is no merge phase.

PostgreSQL can efficiently sort both small and large amounts of data. Small amounts are sorted in-memory using quicksort. Large amounts are sorted using temporary files and a standard external sort described in [3]. In fact all the tuples coming from the children nodes in execution plan are stored in a heap, which has its size limited by the value of user set parameter workmem. When the heap gets completely full, the internal

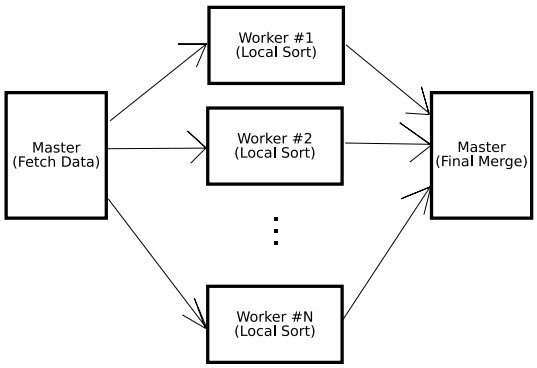


Figure 3.1: Data Flow of Parallel Sort

sorting algorithm is switched to external and all tuples are flushed to temporary files. This algorithm is implemented in the module tuplesort.c. We extended this module to support a parallel merge-all sort, where the data is first sorted locally in workers and then one node performs the final merge phase. In our case this node will be master backend. But at first we have to distribute the data to workers which we have done by using round robin logic to effectively eliminate the skew. In the context of our infrastructure we have also defined a state machine shown in the Figure 3.2, which is used to synchronize master with workers. For example when master has to wait for all workers to finish their local sorting so he can make initial filling of his heap used for final merge phase. The sort structure transferred to worker contains:

- Tuple description - how does the tuple look like.
- Sort column index - index of column on which it should be sorted (one for each sort column).
- Sort operator - for example "lower then or equals" operator (one for each sort column).
- Nulls first - if nulls should be first (one for each sort column).
- Other internally required parameters like scan direction or if it is bounded sort.

To allow parallel sort for all queries, user has to edit these new parameters located in standard PostgreSQL configuration file postgresql.conf:

- prl_sort - if enabled, parallel version of sort will be used (default off).

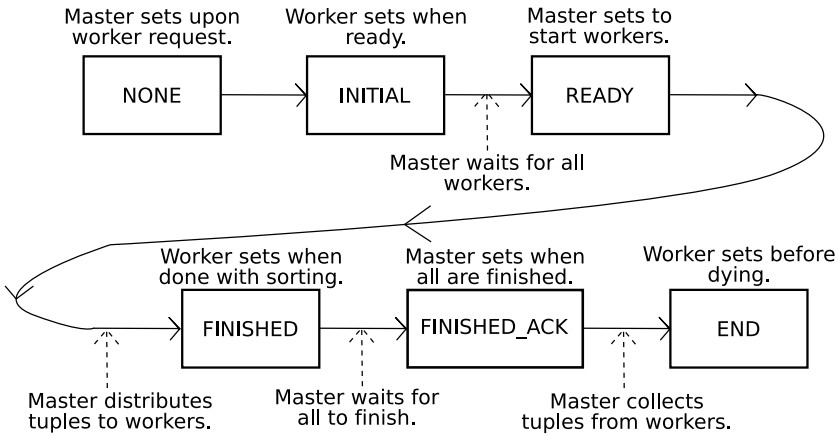


Figure 3.2: Worker states in Sort

- prl_sort_dop - specifies how many workers should be used (default 2).

This implementation is good demonstration of our parallel infrastructure but it has some problems in terms of performance. It is because it strongly depends on how fast we can transfer all the tuples from master to workers and back. We are also limited by the disk speed on which are all the tuples from external sort temporarily stored. During the testing we have realized, that our shared memory approach is very comfortable to use, but not that good for transferring millions of tuples. This problem and possible solutions will be discussed at the end of this chapter.

Apart of the mentioned shared memory problem we have also identified other possible improvements. E.g., we could store the data in the master to external storage and transfer only the structure which describes them, namely logical tape set. But this structure is very complex and depends at least on PostgreSQL implementation for limited number of open files, which makes transferring this structure even more complicated. Also, if we could specify where should the workers store their temporary files, we could overcome a problem in which multiple processes try to concurrently write huge amounts of data to a single drive.

But while we don't have an implementation for quickly transferring data between processes the performance results are not good.

3.2 Append node

In the last section we have implemented an example of intra-node parallelism but it becomes much more interesting when we start to implement inter node parallelism. I.e., the situation in which we execute multiple plan nodes concurrently. This can be done in many ways, for example we can create identical copies of some part of execution plan and execute each of these copies on a different worker. They execute the same thing, but on a different subset of the data. Another option is to actually split the plan to subtrees and let them being handled by worker backends. This can be beneficial when individual subtrees include materialized nodes. In case of a serial execution which uses depth first pipelined execution we could only materialize one node at a time. Such as sort, which needs to fetch all the tuples prior to outputting first tuple. This could theoretically give us a linear speed up.

The decision of which parallelism form to use might be done by a planner. However, we will implement a simple scenario in which we do not need to change the planner implementation. It is the situation in which the Append Node is responsible for sequential execution of all its sub-plans. This node is used when there is a UNION ALL operator present in SQL query, or in PostgreSQL it also handles table inheritance. Our goal is to execute each of these execution plan subtrees in a separate worker.

As a proof of concept we implemented a version in which we defined the work for workers as a SQL select statement. These queries were defined as standard parameters which masters reads and forwards to workers. They then execute them in their own transaction, i.e., they do their own parsing, planning and execution. To get the results from the workers we have defined a new node, whose sole purpose is to send all the tuples to the master through shared buffer queue. We have named this Node as a PrlSendNode and it is defined in its own module prlSendNode.c. So after the workers construct their plans we have to add this node on the top of the plan. This would have normally be done by a planner.

In the second phase we have built a more useful variant. When the execution in master backend reaches the append node and the parallelism is allowed it decides how many worker backends it will use. Decision is based solely on an actual number of sub-plans and maximal allowed number of workers (set by parameter). Then the master sends his request to the postmaster. Afterwards we copy the sub-plans into the shared memory, specifically into the masters shared memory message context. This would allow us a convenient disposal after the workers are done with

it. We transfer the following attributes to worker backends:

- copyPlan - defines whether we are sending actual plan or just SQL query.
- query_string - SQL query or NULL when sending a plan.
- subnode - tree structure with plan or NULL when sending a query.
- rtable - list of relations which are referenced in a plan or NULL when sending a query.

Similarly as in the first version, we have to add the sending node. After the workers starts to process their sub-plans then master actively checks their output queues in shared memory and sends the results to the client. When the queue is empty, it does not wait as in parallel sort, but rather returns with a predefined result value and checks the next queue. To prevent an excessive CPU usage we suspend the master for a small amount of time when all queues are empty. This behavior of not waiting and checking the next worker has a direct impact on the order in which the results are returned. In serial execution we could expect the data to respect the order of sub queries, but here it can generally be in any order. This is not in a conflict with SQL specification since the UNION ALL is a set (specifically multi-set) operation. However, the users should be aware of this behavior.

User could have specified lower maximal number of workers than the actual number of sub-plans. The execution of some branches of plan tree has to wait until one of the workers has returned all of its results. Then we can reuse it for next unprocessed subtree. When the worker has fetched and sent all the results to the master, he then sends a specially marked record. This way we know if the worker has finished or not, without needing to check his state. However this can be done only when the worker reaches the end of its execution. This might not happen for example when a user specifies the maximum amount of rows by using a LIMIT clause. And considering the random nature in which we fetch rows from sub-plans as they are produced, we cannot know how many tuples will be used from specific workers. This will result in the worker processes going into the sleep state on semaphores after they have filled their shared queues. We handle this situation in the ending routine of the append node in two steps. At first we check if the queue is empty and if not we mark the queue as stopped and remove one element. This will wake up the worker process, which then tries to finish the insertion

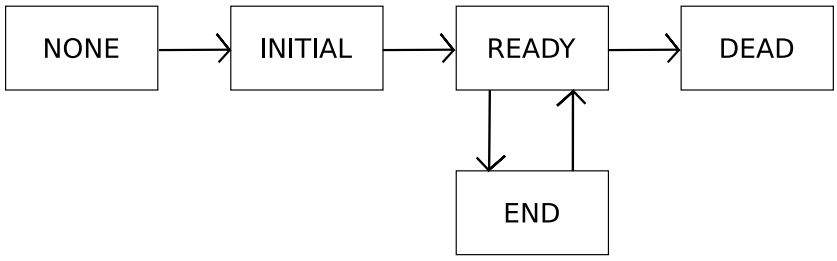


Figure 3.3: Worker states in Append Node

but will notice that the queue is marked as stopped so it will not try to add any more elements. In the second step we terminate the worker backends by sending them a special terminating task.

We have run some tests to see how this implementation stands in terms of a performance. Testing data consisted of tables with two columns. The first column was randomly generated floating point value upon which the aggregation function will be calculated. And the second one was an integer value on which we grouped these rows. Sample testing script looks like:

```
CREATE TABLE TEST1 (val NUMERIC, grp NUMERIC);
CREATE TABLE TEST2 (val NUMERIC, grp NUMERIC);
CREATE OR REPLACE FUNCTION random(numeric, numeric)
RETURNS numeric AS
$$
   SELECT ($1 + ($2 - $1) * random())::numeric;
$$ LANGUAGE 'sql' VOLATILE;
INSERT INTO TEST1
SELECT random(), random(1,3)::int
FROM generate_series(1,10000000);
INSERT INTO TEST2
SELECT random(), random(4,6)::int
FROM generate_series(1,10000000);
SET PRL_SQL = ON;
SET PRL_SQL_LVL = 2;
SELECT SUM(VAL), GRP FROM TEST1 GROUP BY GRP
UNION ALL
SELECT SUM(VAL), GRP FROM TEST2 GROUP BY GRP;
```

With the usage of an aggregation function and group by clause we have effectively reduced the size of a result set. This way we are not limited by the data transfer speed between processes. From the results

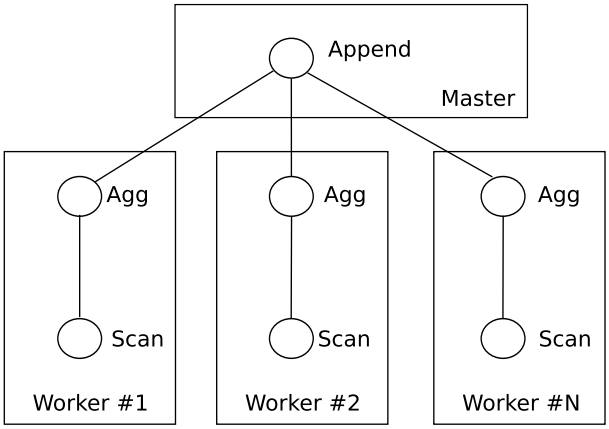


Figure 3.4: Execution plan with workers

Table 3.1: 2 Tables on a single standard HDD (in ms)

Tuples	Prl On			Prl Off		
	Min	Max	Avg	Min	Max	Avg
1k	44	62	51	17	21	19
10k	89	158	124	66	159	111
100k	334	400	368	562	627	586
1M	2825	3474	3018	5229	7144	5384
10M	60400	66967	62716	56974	95047	70548

shown in the Table 3.1 we can see that the bottleneck is the disk storage. As long as the data size was small we could see performance gains but when processing larger tables (hundreds MB) the execution was not getting anything from our parallel implementation. But after creating a separate tablespace on a second hard drive and placing one of the tables we observed the speed up even on large tables. The results are shown in the Table 3.2.

This approach of partitioning data on multiple different physical storages is very common and one of the cornerstones of parallel processing. With the arrival of solid state drives (SSD) and their widespread use we have also performed these tests on a SSD. Results show that these disks outperforms classic HDD and are therefore more suitable for parallel processing. However there are still some problems with their usage in a mission critical systems as their behaviour is not very well document as described in [4].

In this implementation we have shown that the inter-node parallelism

Table 3.2: 2 Tables - standard HDD & USB external (in ms)

Tuples	Prl On			Prl Off		
	Min	Max	Avg	Min	Max	Avg
1k	52	62	57	17	21	19
10k	97	132	115	84	170	115
100k	260	464	385	545	614	579
1M	2838	3688	2986	5205	5287	5239
10M	31041	38930	33380	57513	61438	58554

Table 3.3: 2 Tables - single SSD (in ms)

Tuples	Prl On			Prl Off		
	Min	Max	Avg	Min	Max	Avg
1k	21	33	27	5	15	8
10k	34	60	38	26	39	30
100k	220	268	237	203	297	223
1M	2688	3107	2958	2636	2890	2777
10M	31461	33540	32294	30660	30912	30795

in PostgreSQL is possible and even in simple forms can produce significant speed gains. It highly depends on the queries being run and overall setup and architecture of the database. In the following sections we will discuss this implementation and also suggest further work.

3.3 Discussion

3.3.1 Problems

With almost every decision we had to consider multiple factors and often we could not reach a one hundred percent correct solution. In order to continue in our effort we often had to build workarounds or loosen our requirements. In this section we will mention some problems we have faced and how we have dealt with them.

One problem we were unable to solve properly were transactions. PostgreSQL is fully complaint with ACID SQL specification and it implements it as a multi version concurrent control (mvcc). This is very suitable for complicated situation such as concurrent single row updates. Users can effectively choose from 2 levels of isolation, read committed and serializable. Currently, the transactions can run only inside one back-

end process. But as we have split the execution into multiple processes that each can do for example its own table scans, we had to think about transactions. One possibility is to use sub-transactions, also called nested transactions support, implemented in PostgreSQL as savepoints. Or for example try to keep the same transaction identifier in all participating backends. There are also many other ways how to solve this problems. Let's mention looking at this problem as some kind of distributed transactions. And we also have to consider what can be parallelized and what not. Looking at other RDBMS we can see that updates are actually not supported. Probably because it is very difficult to keep the database in a correct state under all circumstances with multiple execution units trying to do updates or deletes under the same transaction. However with PostgreSQL using mvcc it even might be possible. But to solve this problem it would be necessary to get a very deep knowledge of specific implementation details of transaction support in PostgreSQL. And this is not an easy task to achieve. What we were able to do and considering that we only allow running read only queries in parallel is that each backend is running in its own transaction. In read only queries they should not end up in a deadlock. In parallel sort we actually use the transaction only for internal purposes as opening transactions initializes some structures, such as resource managers and prepares temporary file support for external sort. So the transaction support is a hot candidate for solving in future releases, but quite difficult.

Another rather problematic area is a non propagation of PostgreSQL parameters between processes. Currently, they can be specified globally for all backends in standard configuration files and then locally for currently open session. These locally set parameters are then valid only for this one open connection and lost upon closing. But as we added new worker backends created by forking from postmaster, they are not aware of any changes in parameter values done in master backend. So this can potentially cause problems and therefore has to be solved. One option is to reset parameters in workers before executing each task to respect master settings.

Error management is probably one of the most difficult problems we have encountered. As we have seen in the previous sections, even the basic situation when user decides to cancel the current query is quite difficult to handle correctly. This often leads to cumbersome and not robust solutions. Also quite common problem was even to decide what to do when some error condition happened. For example when we run out of shared memory. We could allocate more memory or we could raise

Table 3.4: Allocation test

Method	1 process (s)	2 processes (s)
malloc	18	47
context alloc	118	1303
ossp malloc	110	1303

a system error and force the backend to die causing the reboot of whole database system. In this case we have decided to raise an error.

During the development of the parallel sort we have seen low performance when sending large amounts of records through our shared memory finite queue. While our structures over shared memory are very comfortable to use, they are not performing well when doing lots of allocations and deallocations simultaneously in multiple processes. Because it is a single shared memory segment, it causes a natural bottleneck. We proved this by performing tests in which we 1000 times allocated and deallocated array of 50000 blocks of 200B. Once in standard memory via malloc, then by using ossp malloc functions and finally by calling palloc which allocates to the PostgreSQL memory context over ossp segment. We have run these test using a single and one pair of processes. In the Table 3.4 we could clearly see how quick these allocations actually are.

As a possible solution we proposed to re-implement the queue to become a cyclic preallocated queue. This way we would eliminate the necessity to do most of allocations and deallocations in the shared memory. But because the memory will have to be controlled by the queue itself it would lead to increase of its critical section. Newly it would contain not only adding to the list but also copying the data. This critical section issue could lead to similar performance problems as the previous implementation. So it is of the most importance to perform further optimizations of transfer structures. Other option might include defining some granularity over the shared memory so the workers don't have to compete for it.

3.3.2 Future

In the previous section we have discussed things that are in some way implemented and require further attention and in this section we will mention things that could be done to further extend our work.

The most natural succession to our work is adding new nodes, which would redistribute and merge data streams based on defined functions. Functions will include standard range based distributions and hash func-

tion distributions, but heir implementation will have to be tailored for PostgreSQL core.

After these nodes are in place we could focus our work on building a parallel planner. This planner will have to decide whether the parallelism is possible and if it could speed up the execution of a specific query. The actual planning could work in two steps. In the first step the planner would build a serial plan. Then if the complexity of a query or the amount of processed rows would have met the requirements for parallel processing, the planner would add parallel nodes and calculate parallel cost.

Besides of adding new nodes, it could be also beneficial to implement new versions of already existing nodes. For example implementing parallel full scan node.

There are also many other areas which could possibly benefit from parallelism. Such as parallel index building or parallel searching of plans in plan space of complex queries. In theory we could be able to either find more optimal plans or find them in less time.

Considering that the speedup and scaleup are the main goals of parallelism the focus should be also on optimizations of all structures and processes. And because PostgreSQL community is always going in a way of preferring more general and provably correct solutions, instead of using shortsighted quick implementations, it will be necessary to precisely define all interfaces and algorithms.

3.3.3 Conclusion

The PostgreSQL currently does not have any parallel query execution support. This, in addition with vague definition of our task in this diploma thesis, gave us a free choice in deciding what and how we want to implement it. We have started with getting acquainted with the source code, which is very extensive and also very complex system. It took as some time, but has given us an important insight into overall architecture, coding style and basic knowledge of some modules, such as handling memory allocations and executor module. We were then able to set our goals and outline multiple paths into achieving them. Some paths were better than others. For example, we could not decide whether to use threads or processes as execution units. Eventually, we have decided to use threads and started to build our implementation. But we have abandoned this thread approach as we have soon identified many modules that needed to be re-implemented only to be thread safe. However, we can not say that the decision to use threads is wrong and that it is not possible

to use threads as execution units. It is just much more complicated and now we thinks that processes are more suitable for PostgreSQL.

Our conclusion is that parallel query processing and generally parallel processing is possible in PostgreSQL database system. We have successfully implemented basic infrastructure consisting of shared memory allocations management and correct creation of worker processes. We have then defined control structures and implemented some parallel operations. In some scenarios, we have even achieved significant speed gains. However, there are still many unanswered questions and unresolved problems. They would require further attention from core community developers if the parallel processing is ever to reach a production-ready state.

Appendix A

Implementation specifics

The most important part of this diploma thesis are source codes. Because it is done upon existing system with many source code files it is shipped in two ways. One as a standard diff file, which contains only changes that we have done and then a complete source code repository, from which can be the system build and used. For versioning of our files we tried many systems but ended up with a Subversion repository hosted on Google dev project web site. We also adopted a vendor branch approach, which is basically keeping a clean copy of core project source codes downloaded regularly from PostgreSQL repository in one branch and then merging of this branch into main developing branch, called trunk. This way we could keep working on almost up to date version of the core without needing to write into it. And also by comparing these two branches we could get the diff file with only our changes. All of this is placed on the attached CD in /src directory.

A.1 Install guide

This is a simplified manual for installing a complete PostgreSQL system with all described parts in this diploma thesis without needing a copy of included CD.

1. Operating System
    - Install Ubuntu - for example ubuntu-10.04.1-desktop-amd64.iso in VMware which is trivial one click install and you may keep your current system.
    - Add following libraries:
        - libreadline5-dev,

  - zlib1g-dev,
  - bison,
  - flex,
  - java,
  - subversion,
  - libsvn-java,
  - ossp-mm dev.

2. Workspace

   - Download Helios(Eclipse IDE) from
     http://www.eclipse.org/downloads/packages/
     eclipse-ide-cc-developers/heliosr.
   - Untar it.
   - After starting it set up workspace.
   - Add subclipse from Help - Install New Software
     URL is http://subclipse.tigris.org/update_1.6.x

3. Source code

   - Switch to "SVN Repositories" perspective in Helios.
   - Add new repository location for
     https://prlpg.googlecode.com/svn/trunk/.
   - Checkout prlpg.
   - Switch to terminal and go to project directory.
   - Run ./configure –prefix=$PROJECT/prlpg –enable-depend
     –enable-cassert –enable-debug
     $PROJECT replace by your project location.
   - Add to LIBS -lmm in Makefile.Global.
   - Build it in Helios.
   - Add "Install" make target and run it - It will create runtimes.

4. Database set up.

   - In terminal go to bin in your project and run following:
     - export PATH=$WORKSPACE/prlpg/bin:$PATH
       $WORKSPACE replace by your workspace location.

  - export PGDATA=$WORKSPACE/data
    $WORKSPACE replace by your workspace location.
  - initdb

5. Run/Debug setup in Helios IDE

   - In Run Configurations - New C++ Application - project prlpg, runtime postgres
     - arguments -D $WORKSPACE/data ... $WORKSPACE replace by your workspace location.
   - In Run Configurations - New C++ Application - project prlpg, runtime psql
     - arguments postgres
   - In Debug Configurations - new C++ Attach to application - prlpg project, postgres runtime

Bibliography

[1] Taniar D., Leung C.H.C, Rahayu W., Goel Sushant: *High-Performance Parallel Database Processing and Grid Databases*, Wiley, USA, 2008

[2] Fritchey Grant: *Dissecting SQL Server Execution Plans*, Simple-Talk Publishing, 2008

[3] Knuth Donald E. *The art of computer programming, Volume 3*, Addison Wesley Longman, 1998

[4] Smith Gregory *PostgreSQL 9.0 High Performance*, Packt Publishing, 2010

[5] PostgreSQL Documentation
http://www.postgresql.org/docs/9.0/interactive/index.html

[6] PostgreSQL Source Code Documentation
http://doxygen.postgresql.org/

[7] OSSP mm home page
http://www.ossp.org/pkg/lib/mm/