Tomograph: Highlighting query parallelism in a multi-core system

Mrunal Gawade
CWI, Amsterdam
mrunal.gawade@cwi.nl

Martin Kersten
CWI, Amsterdam
martin.kersten@cwi.nl

ABSTRACT

Query parallelism improves serial query execution performance by orders of magnitude. Getting optimal performance from an already parallelized query plan is however difficult due to its dependency on run time factors such as correct operator scheduling, memory pressure, disk io performance, and operating system noise. Identifying the exact problems in a parallel query execution is difficult due to inter-dependence of these factors.

In this paper we present *Tomograph*, a tool to visualize the parallel query execution performance bottlenecks. Tomograph provides a time ordered view of operator execution aligned with cpu, memory, and disk IO usage, in an operator at a time execution model. We discuss the usage of Tomograph to identify parallelism issues such as low multi-core utilization, erroneous operator scheduling, incorrect data partitioning, and blocking operators. We share our experiences, insights gained and discuss possible solutions to the identified problems.

Categories and Subject Descriptors

H.4 [**Database Architecture**]: Performance Analysis

General Terms

Query Execution Analysis Tools

Keywords

Visualization, Query Parallelism Bottlenecks

1. INTRODUCTION

Query execution performance analysis tools assume a very important role for tuning database systems [1][2]. Some of the common methods for performance analysis are *explain plan* based operator statistics analysis [3][4], profiler based SQL query analysis [5], and query execution trace based fine grained analysis [6]. Explain based analysis often involves a visualization of query plans in a static graph format, annotated with operator statistics. Execution trace based analysis involves filtering of trace attributes and front end visualizers which aggregate trace information to provide a condensed view [7]. The volume of information contained in a trace, both in terms of number of attributes and quantity of information, could grow very large. The query performance analysis using the methods described before are well suited for SQL / operator statistics based analysis. However, parallel query execution analysis on the basis of operator time ordered issues is difficult using these methods.

Hence, time ordering based visualization methods assume unusual importance in the world of parallelism. In a parallel execution, time ordering of parallel execution context provides a lot of insight about the state of the system. Identifying problems such as computational skew in parallel threads is easy by visualizing the execution timeline of each thread side by side, as compared to deciphering it from statistics of individual operators in a graph visualization. Problems such as wrong operator scheduling in a complex plan which are very difficult to pinpoint otherwise can be spotted quickly by visualizing the operator execution ordering. Getting an insight about the possible parallelism problems is thus a matter of coming up with a correct visual scheme.

In this paper we introduce a new visual tool the *Tomograph*, that helps in identifying performance bottlenecks in a parallel query execution, on a single canvas. Tomograph provides a coherent time ordered view of operator execution, cpu usage, memory usage, and disk IO activity. It improves the ability to pinpoint the performance issues during parallel query execution. We use it to explore the operator at a time execution model of the MonetDB execution engine [8].

Paper structure: The structure of the paper is as follows. In Section 2 we provide a brief introduction to the existing parallelism techniques. We also discuss the operator at a time execution model and the parallelism technique it uses. Section 3 elaborates on the Tomograph architecture. In Section 4 we illustrate the analysis of different parallel query execution bottlenecks using Tomograph. Section 5 discusses the related work. We conclude in Section 6.

2. QUERY PARALLELISM

This section provides a brief introduction to the existing query parallelism techniques. It also discusses parallelism in the context of the operator at a time execution model.

Query parallelism techniques could be broadly classified on the basis of granularity of parallelism [9]. The classification at the highest granularity could be as inter-query and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WOODSTOCK '97 El Paso, Texas USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

```
X_3  := sql.bind(X_2,"sys","table","a",0);
X_17 := algebra.uselect(X_3, A1, A2);
```

Figure 1: A serial MonetDB Assembly Language (MAL) plan with a single select operator

intra-query parallelism [10]. In inter query parallelism two or more queries execute in parallel, while sharing available resources such as CPU, memory, and disk IO. In intra-query parallelism, multiple operators in a query plan execute in parallel. Intra-query parallelism could be further divided into inter-operator and intra-operator parallelism. In inter-operator parallelism, two or more disjoint operators execute in parallel. In intra-operator parallelism multiple instances of the same operator operate on range partitioned data. Operators in a serial plan of a pipelined query execution model (open-next-close) exhibit inter-operator parallelism, whereas in a parallel plan the operators parallelized using the exchange operator mechanism exhibit intra-operator parallelism [11].

This paper uses an operator at a time execution model for elaborating the use case of Tomograph. In the operator at a time execution model, an operator executes completely before the next data flow dependent operator execution begins. We describe this execution model in the context of MonetDB, the open source columnar database system. MonetDB operators are represented in MonetDB Assembly Language(MAL) [12]. MAL operators are mapped to their relational algebra counterparts. In Figure 1 *X_17 := algebra.uselect(X_3, A1, A2);* is a MonetDB instruction. The uselect operator in this instruction represents the relational algebra operator scan select. MonetDB uses binary association tables (BAT) to store the result of an operator execution. In the instruction X_17 represents a BAT which stores the result of the operator uselect. Uselect accepts X_3 as an input BAT and A1, A2 as the range select parameters. Uselect executes completely producing the intermediate result in the variable X_17 justifying the name operator at a time execution model.

The operator at a time execution model exhibits both intra-operator and inter-operator parallelism. Consider the parallel plan in Figure 2. It is obtained by 2 way partitioning of the column *a* from the serial plan in Figure 1. Two instances of the *uselect* operator operate on each of the partitioned column ranges. Another operator *mat.pack* combines the output of the two uselect operators in the end. Let us name each of the 6 instructions in the parallel plan in Figure 2 as A, B, C, D, E, and F. Instruction C is data flow dependent on the output of instruction A (A->C). Data flow dependency in other instructions are (B->D) and (C,D->E). Hence, the instructions (A,B) and (C,D) could be executed in parallel. Instructions C and D executing in parallel is a case of intra-operator parallelism, since the same operator instance is operating on two disjoint range of partitioned data. The instruction F can execute in parallel with any other instruction. Hence, it is an example of the inter-operator parallelism.

2.1 Operator mapping

The main operators in MonetDB could be mapped to their relational algebra operator semantics such as scan, join, aggregation, group, sort, and projection. Being a column store

```
A  X_3 := sql.bind(X_2,"sys","table","a",0,1,2);
B  X_4 := sql.bind(X_2,"sys","table","a",0,2,2);
C  X_17:= algebra.uselect(X_3, A1, A2);
D  X_18:= algebra.uselect(X_4, A1, A2);
E  X_19:= mat.pack(X_17, X_18);
F  X_20:= alarm.time();
```

Figure 2: A parallel MonetDB Assembly Language (MAL) plan with 2 way partitioned select operator

there are also many administrative operators which are used for tuple reconstruction. However, in this paper we focus only on the main operators. The main operator mapping is as follows.
1. Select - algebra.uselect, algebra.thetauselect, algebra.slice
2. Join - algebra.join, algebra.leftjoin, algebra.semijoin
3. Aggregation - aggr.sum, aggr.count, aggr.groupby, mat.pack
4. Arithmetic- batcalc.-, batcalc.*
5. Groupby- group.multicolumn, group.sort, group.done

Based on the query type, the amount of partitioning, and the operation context, the execution duration of operators could vary considerably. The main operators execute for a long duration (seconds or ms) compared to the execution duration of the administrative operators (μs). Administrative operator's execution can often not be visualized as its duration granularity is too small to display, as compared to the total query execution time. Hence, Tomograph displays only the expensive operators.

3. TOMOGRAPH

Tomograph is a visual tool that provides a time ordered view of operator execution in alignment with the multi-core CPU, the memory, and the IO activity. It helps in understanding where does the time go during parallel query execution from individual operator execution perspective. In this section we describe in brief the architecture of Tomograph. We also provide details about how to read a Tomograph.

3.1 Tomograph architecture

Tomograph is a command line client which connects to a MonetDB server. It uses GNUPlot to visualize the MonetDB execution trace. Tomograph is implemented in "C".

Tomograph uses a UDP connection to connect to the MonetDB server. When a query is fired from a MonetDB client, the MonetDB profiler starts to send the execution trace to Tomograph on the UDP connection. The trace consists of information such as operator start time, finish time, total time of execution, thread id, cpu usage, memory usage, io usage, etc. Tomograph parses and filters out the trace information to generate input data points for different type of graph generation. Once the complete profiled data is available the visualization phase begins.

Tomograph creates a GNUPlot script to generate multiplots from the available profiled attribute data. Each operator in the main graph is color coded. The color code map is static to help in the comparison of the same operators across different query invocations. A single query plan can contain a large number of operators making it hard to understand the main graph. With multiple graphs on a single canvas and a lot of information on the display, an understanding of how to read the Tomograph is essential. We describe it in

the next subsection.

3.2 How to read a Tomograph

We explain how to read a Tomograph using the graph from Figure 3 as an example. The main graph contains operator execution timeline and should be read in alignment with the graphs containing information on the CPU, the memory, and the IO activity.

The operator analytics information for the main graph is provided by the legend present at the bottom of the graph. The legend provides a color map for each operator in the plan along with the aggregated execution time for all operators of the same type. Each operator has a static color. The operators are classified as expensive or cheap based on their duration of execution. The duration of operator execution could vary from a few microseconds upto seconds. The main graph displays only the expensive operators. For example, from Figure 3 consider the operator *algebra.leftjoin* (yellow). There are 82 such operators. The total aggregated time of execution of all these operators is 5.26 seconds. Since this time is considerable as compared to the total execution time, this operator is labelled as an expensive operator. However, the distribution of execution duration amongst 82 operators could vary. As long as there is a single operator with considerable duration, the operator is categorized as an expensive operator.

The main graph displays the operator execution timeline on multiple threads of execution. Each horizontal line represents a thread of execution. The total number of lines equals the total number of worker threads. Each thread executes multiple operators sequentially. The presence of a colored box on a thread line indicates an operator under execution. The length of the box represents the duration of operator execution. (The language.dataflow operator on thread 4 should be ignored in all graphs as it's a proxy operator). An operator execution representation could indicate multiple possible system states such as an actual computation in progress, wait on lock, cache, memory, or disk IO access. Tomograph does not deal with the granularity of dissection of what happens during individual operator executions. The presence of a white space on the thread line indicates the lack of operator execution. The smaller the white space, the better the multi-core utilization of the system.

The CPU activity per core is displayed in the graph immediately above the main graph. Each horizontal line indicates one core. The line oscillates depending on whether an operator is being executed or not.

The resident set usage graph is present above the CPU activity graph. It shows the memory consumption by the MonetDB server. The memory usage grows and shrinks on the basis of the intermediate data generation. This graph also represents disk IO activity in terms of read and write IO's per miliseconds, if IO occurs.

4. EVALUATION

Tomograph is being used for analysing parallel query execution performance problems. In this section we describe our analysis of some of the identified problems such as inefficient multi-core utilization, incorrect operator scheduling, data partitioning, blocking operator, and static heuristic rules. We use TPC-H benchmark queries on the scale factor 10 dataset for the evaluation. All queries are executed during

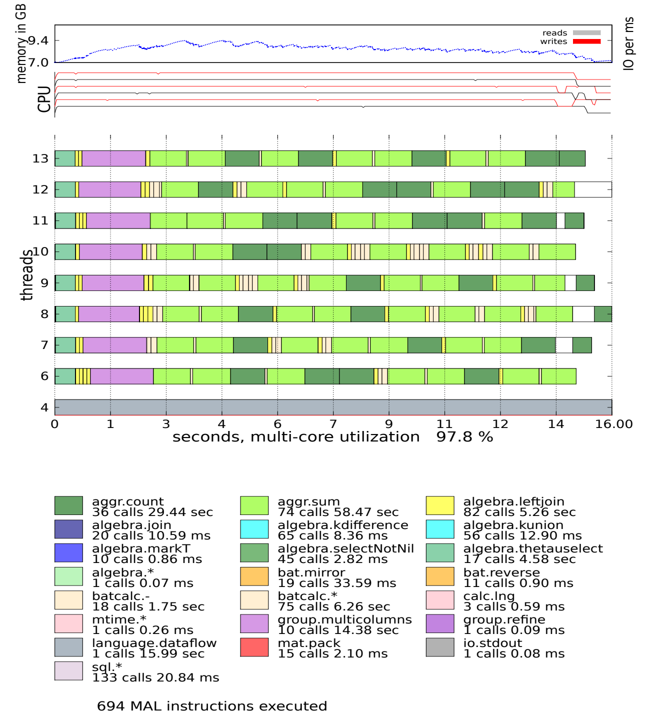

Figure 3: TPC-H Q1 hot run execution on scale factor 10 shows excellent multi-core utilization.

hot runs of the MonetDB server (default branch changeset c56e636745dd). The operating system Fedora 16 operates on Intel Core i7-2600 CPU @ 3.40GHz, 16 GB DDR3 Dual channel RAM, and a 7200 RPM 1 TB SATA hard disk.

4.1 Multi-core utilization

We define multi-core utilization as the sum of the execution time of all operators on all active threads divided by the total time of all active threads. Higher multi-core utilization ensures optimal resource usage. Next we analyze queries 1 and 10 from Figures 3 and 4 on the basis of their multi-core utilization.

Query 1 (Q1) represented in Figure 3 shows excellent multi-core utilization at 97.8%. This is evident from the lack of white space on any of the thread lines during the query execution timeline. The maximum multi-core utilization is a result of a simple query plan, which does not have operators such as join. The only table present is *lineitem*, whose column partitions are operated upon by the range select operators.

In comparison Query 10 (Q 10) represented in Figure 4 shows less multi-core utilization at 65.2%. The presence of white space on thread execution lines is a visual indication of less multi-core utilization. The query has selection predicates working on columns of the *lineitem* table and the *orders* table. The query also involves join predicates which makes the plan more complex as compared to Q1. Join predicates are data flow dependent on the result of the selection predicates. This introduces a waiting period amongst operators leading to less multi-core utilization. We analyse this

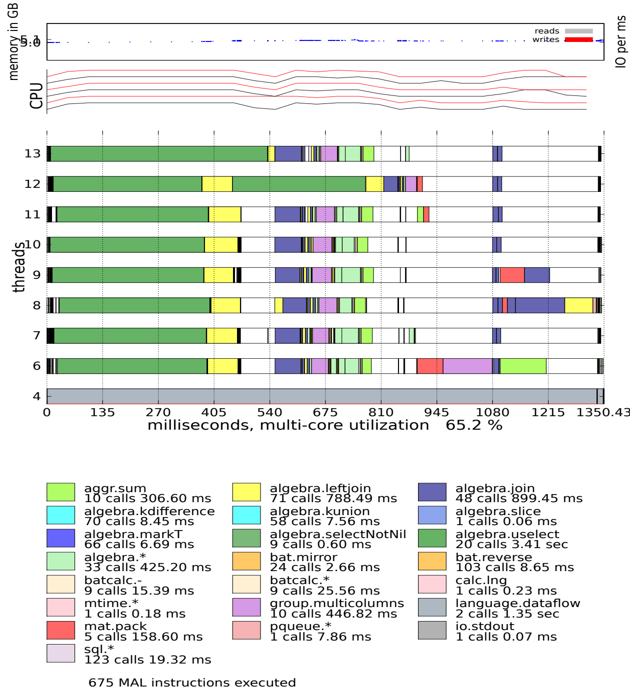

Figure 4: TPC-H Q10 hot run execution on scale factor 10 shows less multi-core utilization as compared to Q1 in Figure 3.

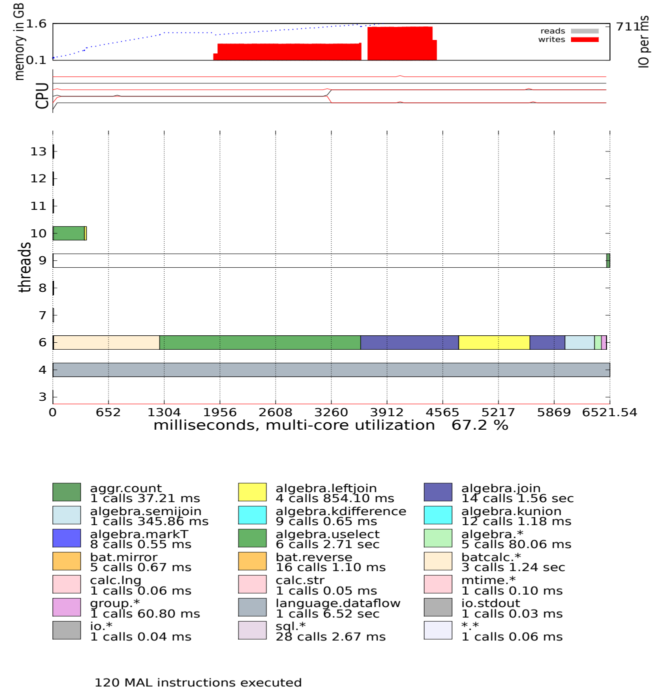

Figure 5: TPC-H Q4 hot run execution on scale factor 10 shows poor parallelism as compared to Q1, Q10 in Figures 3, 4.

behaviour in the next section. Less multi-core utilization is also a result of the presence of dataflow dependent operators such as group.multicolumn (pink). It leads to a blocking behaviour resulting in less multi-core utilization, which we describe in Section 4.3.

4.2 Scheduling and partitioning

Q10 in Figure 4 is a good example of parallel execution problems in operator scheduling and data partitions.

The query has one scan select operation on the lineitem and the orders table column each (green). MonetDB parallel plan re-writer module always partitions columns in the *largest* table in the plan, ignoring the smaller table columns. This rule is in accordance with the OLAP setting, where the fact table is the largest table (hence partitioned), and dimension tables are smaller (hence replicated). The presence of 8 CPU cores leads to 8 equi-range disjoint partitions of the lineitem table column. One *uselect* operator works on each partition. A single *uselect* operator works on the orders table column, since it is not partitioned.

The scheduler schedules 8 execution threads. It has two possible choices. 1. To schedule 8 uselect operators on the 8 way partitioned column. 2. To schedule 1 uselect operator on the non-partitioned single column and 7 uselect operators on 8 way partitioned column. In Figure 4 we observe that the scheduler uses the 2nd choice by scheduling a single uselect operator (topmost left green) on thread 13 and 7 partitioned uselect operators (green) on threads 6 to 12. The 8th uselect operator gets scheduled on thread 12 after the first uselect operator scheduled on it finishes its execution. The uselect operators on threads 6 to 11 finish first. However, they continue to wait till the uselect operation on the non-partitioned column (thread 13) finishes. This wait introduces idle time on multi-cores. The partitioned join operator which is scheduled next (purple) needs results from both types of (non-partitioned column and partitioned column) uselect operators as its input. The join operators following the 6 uselect operators (thread 6 to 11) start execution as soon as the uselect operator on thread 13 finishes.

This case shows the problem of incorrect scheduling order and incorrect number of partitions. One possible way to resolve this issue is by identifying the correct number of partitions. For example instead of 8 partitions of the lineitem table column, use only 7 partitions. This would avoid waiting time for other threads, as they wait for the 8th select operation to finish. The other possibility is to parallelize the select operation on the single non-partitioned column of the orders table and schedule it to execute first. This would serialize the execution of select operations on the orders and the lineitem table, thereby removing the waiting time.

4.3 Blocking operator

A blocking operator in an operator at a time execution context is an operator which does not allow any other operator to execute during its execution. The blocking behaviour is a result of the data flow dependent nature of operators, where a certain operator (blocking operator) accepts as input other dataflow operators output. Group.multicolumn (pink) is a blocking operator in Q10 in Figure 4. However, any operator could show blocking behaviour depending on

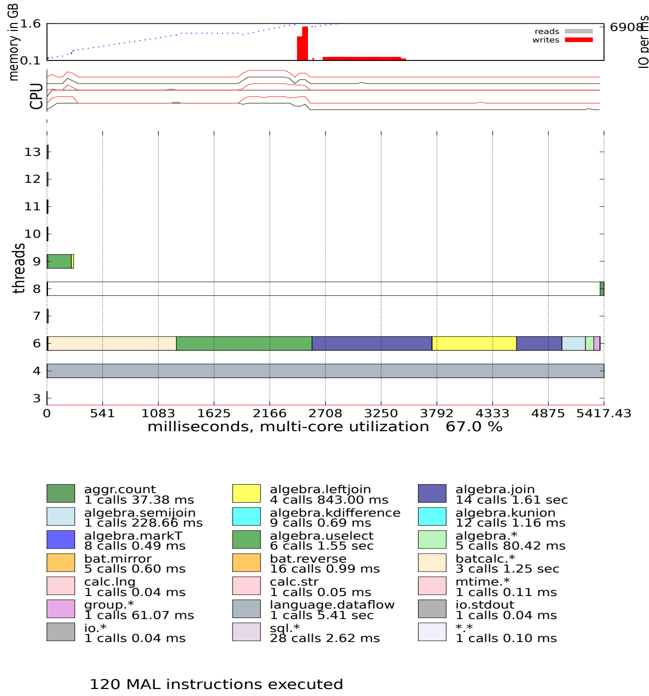

Figure 6: TPC-H Q4 hot execution on scale factor 10 with the intra-operator *uselect*, visible on CPU activity graph. The *uselect* operator execution time shows 41% improvement as compared to the *uselect* operator execution time in Q4 in Figure 5

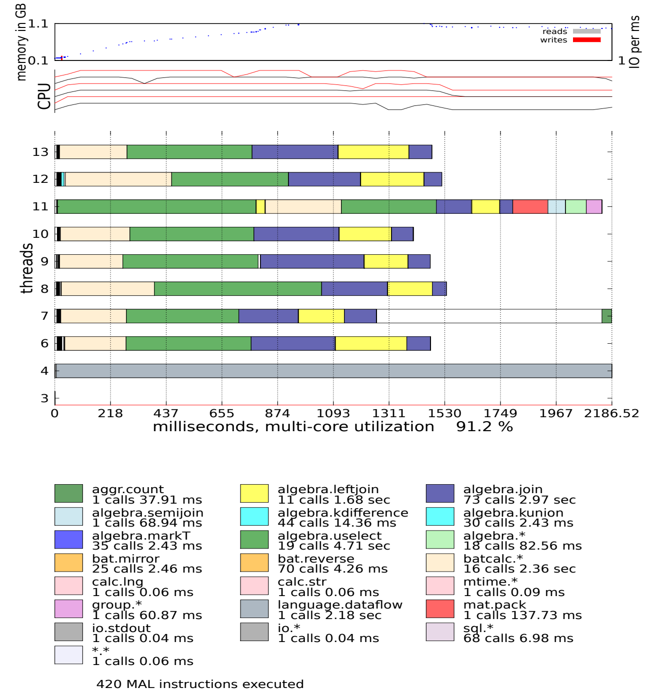

Figure 7: TPC-H Q4 hot execution run on scale factor 10 with forced parallelization. Q4 execution time improves 3 times the Q4 execution time in Figure 5

the type of query. One way to improve the query execution performance in a blocking operator case is to parallelize the blocking operator itself. We describe the experiment of an intra-operator parallel implementation of the operator *uselect* next.

The uselect operator from Q4 in Figure 5 is considered for the intra-operator parallel implementation. We choose the uselect operator as it is a relatively easy operator to experiment with. We choose Q4 as it does not exhibit good parallelism. The presence of only two active threads at the end of the query execution shows that the query is not parallelised. We expect to improve Q4 execution time by intra-operator parallelization of the uselect operator.

Our intra-operator parallel implementation of the uselect operator uses a map-reduce style partitioning of the uselect operator code by spawning *8* threads which do a selection on range partitioned data [13]. The data selected by individual threads is combined by a reducer thread.

The graph in Figure 6 shows the Q4 execution timeline with the intra-operator parallel uselect implementation. *It is visible on the CPU graph where 8 lines are active during the uselect operator execution.* The uselect operator's execution time has reduced to 1.3 seconds from 2.28 seconds, which is an improvement of around 41%. However, we expect at least a 4 times improvement due to the presence of 4 physical cores. A fine grained analysis shows the time for the mapper and reducer phases as 550 and 80 miliseconds respectively. We also observe an interesting fact that the parallel activity for 8 cores in the CPU activity graph does not span the entire uselect execution time. These types of discrepancies are possible to notice immediately due to Tomograph's coherent resource utilization visualization ability.

4.4 Static heuristic parallelization rules

Query 4 in Figure 5 exhibits almost no parallelization. It however shows good multi-core utilization (as defined in Section 4.1) as only two active threads are present during the query execution. The poor parallelism is a result of a non-optimal partitioned plan. MonetDB uses a heuristic rule based plan rewriter module to generate a parallel plan from a serial plan. During the plan rewriting phase, the module suspects the problem of plan explosion. Plan explosion results when the number of instructions in a parallel plan increases to a large number. In Q4 the problem of *join* explosion arises. The plan re-writer module introduces a cartesian product of join operators. The join operator, depending on the type of data (unique, random) it operates on, does different amounts of work. As an example if there are multiple repeated values in the two columns being joined then the result is huge. The heuristic rule based plan re-writer checks for such conditions. If it suspects matching conditions, the parallel plan generation is suspended. Q4 does not get parallelized due to this condition.

In this experiment we force the parallel plan rewriter-module to generate the parallel plan to understand the severity of the plan explosion case. The heuristic rule condition which checks for the join condition for Q4 is deactivated. The resultant exploded plan visualized in Figure 7 contains

420 instructions, as compared to the 120 instructions in the plan visualized in Figure 5. The expanded plan contains 60 instructions with a join operator. However, the query execution time improves by 3 times as compared to the Q4 execution time in Figure 5. This is a result of increased parallelism and 91.2% multi-core utilization. The execution time of individual operators decreases by more than 4 times. Query parallelization is beneficial if the parallel query execution time is considerably less than the serial execution time. If there are too many instructions in the parallel plan it might incur overheads in terms of instruction interpretation, scheduling, result combination, etc. These overheads might negate the gains due to parallelism. This experiment shows that static heuristic rules could misjudge parallelism decisions and could hamper query execution performance.

5. RELATED WORK

Most database systems use the *exchange operator* based parallelism. The exchange operator was introduced by the Volcano system [11]. It is an auxiliary operator which is introduced in a serial query plan by a plan re-writer module, depending on the degree of parallelism needed.

Microsoft SQL server uses the exchange operator based query parallelization [14]. It is also being used by Vectorwise, a leading OLAP system [4]. Postgres uses a pool of database servers to have maximum multi-core utilization [15]. ORACLE uses a technique based on distributed query processing [16]. DB2 uses a just in time plan parallelization approach based on dynamic resource availability [17]. High performance analytic appliances by Netezza use a distributed query processing approach using a combination of FPGA based accelerators and multi-core CPUs [18].

Most of these systems use a legacy command *explain* to visualize the query plan in a tree based format, with added functionality such as operator statistics and color coding [1]. Microsoft SQL server uses a GUI based suite of tools named the SQL server management studio [19]. Vectorwise uses graph based plan visualization with node color coding using an open source package, graphviz. Postgres uses similar tools [20]. MonetDB uses a tool called *stethoscope*, to visualize a data flow dependency graph of a query plan [21]. Vtune analyzer from Intel is used for profiling program execution using a similar visualization scheme as Tomograph [22]. However, it does not provide a parallel operator execution analysis the way Tomograph does. Operator execution visualization in a parallel manner as displayed by Tomograph is thus unique.

6. CONCLUSION

Identifying parallel query execution performance bottlenecks holds a crucial importance for getting optimal system performance. In this paper we have presented *Tomograph*, a visual tool that has proved immensely helpful in identifying parallel query execution issues such as low multi-core utilization, incorrect operator scheduling, incorrect data partitioning, and blocking operators. The coherent visualization of the system state in terms of operator execution timeline, CPU, memory, and IO activity helps to reason about each of the bottleneck issues in a holistic manner.

Tomograph benefits from the operator at a time execution model which allows operator execution ordering to be visualized in a discrete manner. However, the underlying visualization principles could prove immensely helpful for other execution models to identify parallel query execution performance bottlenecks.

7. REFERENCES

[1] Dbtools. http://www.en.wikipedia.org/wiki/Comparison_of_database_tools.

[2] Dennis Elliott Shasha and Philippe Bonnet. *Database tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann, 2003.

[3] Immanuel Chan. Oracle database performance tuning guide, 11g release 1 (11.1) b28274-02.

[4] Actian vectorwise technical white paper. http://www.actian.com/media/whitepapers/unsorted/ivw-technical-wp.pdf.

[5] Sajal Dam and Grant Fritchey. *SQL Server 2008 Query Performance Tuning Distilled*. Apress, 2009.

[6] Monetdb profiler. http://www.monetdb.org/Documentation/Manuals/MonetDB/Profiler.

[7] Pramukh Narayan Rao Jadhav. Performance evaluation of oracle parallel execution. Master's thesis, California State University, Sacramento, 2011.

[8] Peter Boncz et al. Database architecture optimized for the new bottleneck: Memory access. In *Proc of VLDB*, pages 54–65, 1999.

[9] M.T. Özsu and P. Valduriez. Distributed and parallel database systems. *ACM Computing Surveys (CSUR)*, 28(1):125–128, 1996.

[10] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[11] Goetz Graefe. *Encapsulation of parallelism in the Volcano query processing system*, volume 19. 1990.

[12] Monetdb assembly language. http://www.monetdb.org/Documentation/Manuals/MonetDB/MALreference.

[13] Colby Ranger et al. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, pages 13–24, 2007.

[14] Sql server parallelization. http://www.simple-talk.com/sql/learn-sql-server/understanding-and-using-parallelism-in-sql-server/.

[15] Postgres-pgpool. http://www.pgpool.net/mediawiki/index.php/Main_Page.

[16] Oracle parallelization. http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-parallel-execution-fundamentals-133639.pdf.

[17] Y. Wang. Db2 query parallelism: Staging and implementation.

[18] Stephen C Helmreich and Jim R Cowie. Data-centric computing with the netezza architecture. 2006.

[19] Michael Coles. *Pro T-SQL 2008 Programmers Guide*. Springer.

[20] postgresvisualizer. http://www.dbplanview.com.

[21] M. Gawade and M. Kersten. Stethoscope: a platform for interactive visual analysis of query execution plans. *Proc of VLDB*, 5(12):1926–1929, 2012.

[22] James Reinders. *VTune performance analyzer essentials*. Intel Press, 2005.