DoomDB - Kill the Query

Carsten Binnig, Abdallah Salama, Erfan Zamanian
Baden-Wuerttemberg Cooperative State University
Mannheim, Germany

ABSTRACT

Typically, fault-tolerance in parallel database systems is handled by restarting a query completely when a node failure happens. However, when deploying a parallel database on a cluster of commodity machines or on IaaS offerings such as Amazon's Spot Instances, node failures are a common case. This requires a more fine-granular fault-tolerance scheme. Therefore, most recent parallel data management platforms such as Hadoop or Shark use a fine-grained fault-tolerance scheme, which materializes all intermediate results in order to be able to recover from mid-query faults. While such a fine-grained fault-tolerance scheme is able to efficiently handle node failures for complex and long-running queries, it is not optimal for short-running latency-sensitive queries since the additional costs for materialization often outweigh the costs for actually executing the query.

In this demo, we showcase our novel cost-based fault-tolerance scheme in *XDB*. It selects, which intermediate results to materialize such that the overall query runtime is minimized in the presence of node failures. For the demonstration, we present a computer game called *DoomDB*. *DoomDB* is designed as an ego-shooter game with the goal of killing nodes in an *XDB* database cluster and thus prevent a given query to produce its final result in a given time frame. One interesting use-case of *DoomDB* is to use it for (crowdsourcing) the testing activities of *XDB*.

1. INTRODUCTION

Modern parallel database systems such as SAP HANA [1], Greenplum [6] and Terradata [4] are major platforms for analyzing large amounts of structured data efficiently. Most existing parallel database systems have been designed to run on clusters with highly available hardware components where node failures can be ignored. Thus, fault-tolerance is typically handled in a coarse-grained manner by restarting a query completely when a node failure occurs.

Today, there is a growing trend to build systems for analyzing large amounts of data that run on clusters of commodity machines or on IaaS offerings such as Amazon's Spot Instances where node failures are a common case. Therefore, more recent parallel data management platforms such as Hadoop [7], Impala [3] or Shark [8] use a fine-grained fault-tolerance scheme, which materializes intermediate results to being able to recover from mid-query faults. For example, many of these systems compile queries into MapReduce-based execution plans and typically materialize the output of each map and reduce function (either in memory or on disk).

While such a fine-grained fault-tolerance scheme is able to efficiently handle node failures for complex and long-running queries, such a scheme is not efficient for short-running latency-sensitive queries since the additional costs for materialization often outweigh the costs for actually executing the query. Moreover, adding the materialization costs to the runtime of short-running queries has an additional negative effect since this increases the probability that a node failure occurs during the execution time.

In order to address this problem, we are currently building an open source parallel database system, called *XDB*.[1] *XDB* is implemented using a middleware approach on top of an existing single node database system (MySQL in our case) and offers a cost-based fault-tolerance scheme to restart queries from mid-query faults. Compared to existing systems like Hadoop, *XDB* does not materialize all intermediate results but selects a subset of intermediate results to be materialized (called materialization configuration further on). The main goal of the cost model is to find an optimal materialization configuration for a given query, such that the run-time of a given query is minimized under the presence of node failures.

In this demo, we present our novel cost-based fault-tolerance scheme in *XDB* by showcasing a computer game called *DoomDB*. *DoomDB* is designed as an ego-shooter game and the goal is to kill nodes in an *XDB* database cluster before a given analytical SQL query can produce its complete final result in a given time frame. In the game, database nodes are represented as boxes that need to be destroyed by the user. If a box is destroyed, a node failure is introduced (i.e., the database node is forced to stop working), which triggers *XDB* to redeploy sub-plans executed on that node. The progress of the query execution is displayed in *DoomDB* while the game is running. If *XDB* can not finish the query in a given time frame, the user wins the game. For showcasing different scenarios *DoomDB* takes different input parameters such as a pre-partitioned database and a user-defined SQL query. One interesting use-case of *DoomDB* is to use it for (crowdsourcing) the testing activities of our distributed and parallel database system *XDB*.

2. XDB OVERVIEW

XDB is built using a middleware approach, which leverages an existing single node database for query processing. While the middleware implements our novel concept for the cost-based fault-tolerance scheme (amongst other concepts for data partitioning and parallel query execution), the single node database systems are used

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2594525

[1] `https://code.google.com/p/xdb/`

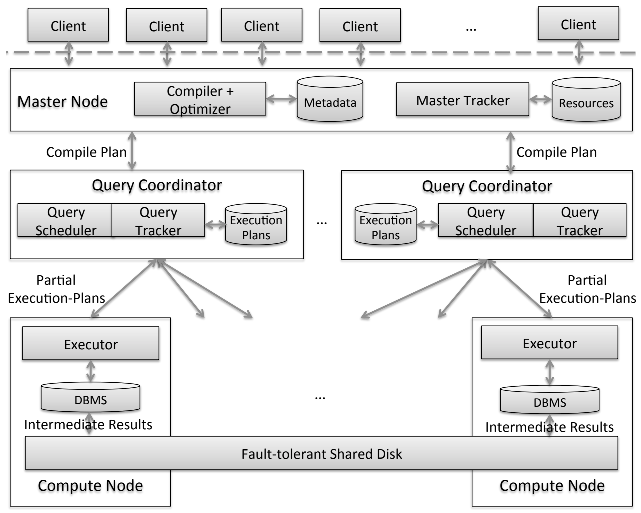


Figure 1: System Architecture of XDB

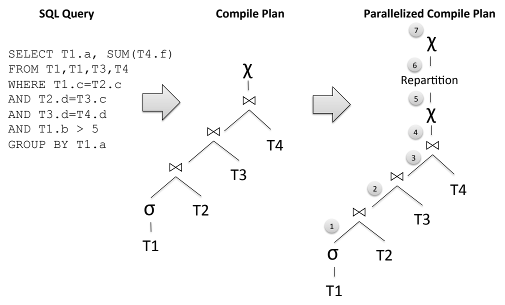
to efficiently store and query the data.

Figure 1 shows the system architecture of *XDB*. An *XDB* cluster consists of one *Master Node*, which accepts analytical SQL queries from clients, several *Query Coordinators*, which are responsible to coordinate and monitor the query execution and finally a huge number of *Compute Nodes*, which actually execute sub-plans of the given query over the partitioned data. In the following, we briefly discuss the tasks of each node type. Details about *XDB* are given in the technical report [2].

Master Node: The Master Node hosts two components, a *Compiler+Optimizer* and a *Master Tracker*. The *Compiler+Optimizer* component takes a SQL query as input and produces a compile plan using the catalog (which holds the table metadata as well statistics). The compile plan is then optimized using rule-based and cost-based optimizations for join-reordering and predicate push-down. The second component, the *Master Tracker*, has different tasks: First, it monitors the cluster resources (Query Coordinators and Compute Nodes) and restarts them when these nodes fail. Second, the Master Tracker assigns a compile plan to a Query Coordinator, which is responsible for coordinating and monitoring its execution.

Query Coordinator: A Query Coordinator hosts two components: a *Query Scheduler* and a *Query Tracker*. The *Query Scheduler* takes a compile plan from the Master Node and splits it into multiple partial execution plans and sends those plans to different Compute Nodes for parallel execution. The result of each partial execution plan is materialized to a fault-tolerant shared disk. Materialization is thus used for implementing our cost-based fault-tolerance scheme, but also for other reasons such as repartitioning data for parallel query processing. The second component, the *Query Tracker*, is responsible for monitoring and redeploying a partial execution plan in case of a Compute Node failure. We separate the Query Coordinator from the Master Tracker to be able to scale the number of Query Coordinators independently from the load in the system.

Compute Node: As discussed before, a Compute Node receives a partial execution plan (represented as SQL-queries) from the Query Coordinator and the *Executor* component then runs these plans. Moreover, the Executor signals the Query Coordinator once a partial execution plan has finished.



```
SELECT T1.a, SUM(T4.f)
FROM T1,T1,T3,T4
WHERE T1.c=T2.c
AND T2.d=T3.c
AND T3.d=T4.d
AND T1.b > 5
GROUP BY T1.a
```

Figure 2: Compilation Process in XDB

3. COST-BASED FAULT-TOLERANCE

In this section, we first present the strategy of how *XDB* enumerates different materialization configurations for a given analytical SQL query. We then present the cost function, which is used to estimate the total runtime resulting for a given query and a materialization configuration.

3.1 Plan Enumeration

XDB uses a two-step approach for enumerating different materialization configurations. It first compiles an optimized parallel compile plan for a given analytical SQL query and then enumerates all possible materialization configurations for the compile plan.

Figure 2 represents an overview of the compilation process: (1) A given analytical SQL query is first compiled and optimized. The result is a left-deep *Compile Plan*, which consists of relational operators. (2) For parallelization, the compile plan is annotated with additional repartition operations that are required when executing the query over a horizontally partitioned database. The result is called a *Parallelized Compile Plan*.

For the resulting *Parallelized Compile Plan*, all the different materialization configurations are enumerated. A materialization configuration defines for each intermediate result whether it should be materialized or not.

For example, in Figure 2 it defines for each of the intermediate results $1-7$ whether it should be materialized. *XDB* currently uses an exhaustive search to enumerate all possible materialization combinations. In general this results in 2^n different combinations for n intermediate results. In its current version, the repartitioning operator of *XDB* always materializes its result. Thus, in our example, we only need to enumerate 2^6 different materialization configurations since the intermediate result 6 is materialized in all cases due to repartitioning.

For each materialization configuration, *XDB* invokes its cost function (described in the next section) to estimate the total runtime under failures and selects the materialization configuration with the minimal estimated total runtime.

3.2 Cost Function

The goal of the cost function is to estimate the runtime for a given *Parallelized Compile Plan* and a given materialization configuration.

Input: For estimating the total runtime under failures, the cost function takes the following parameters as input (1) an *Abstract Execution Plan*, which is derived from the Parallelized Compile Plan, (2) the runtime of each operator for a given materialization configuration (3) the mean-time-between-failures (MTBF) of the Compute Nodes, and (4) the mean-time-to-repair (MTTR).[2]

[2]We assume all Compute Nodes have the same MTBF and MTTR.

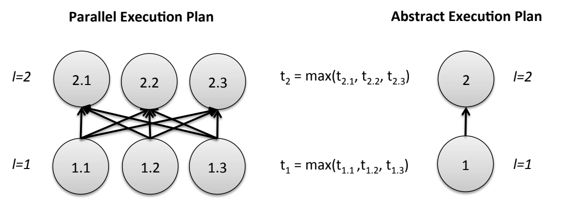

Parallel Execution Plan
Abstract Execution Plan
l=2
2.1
2.2
2.3
$t_2 = \max(t_{2.1}, t_{2.2}, t_{2.3})$
2
l=2
l=1
1.1
1.2
1.3
$t_1 = \max(t_{1.1}, t_{1.2}, t_{1.3})$
1
l=1

Figure 3: Parallel Execution Plans in XDB

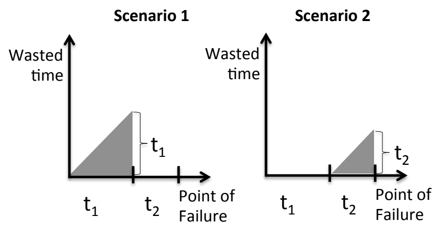


Figure 4: Wasted Time Model in XDB

The first input (1) is an Abstract Execution Plan, which is an abstract version of the actual *Parallel Execution Plan* in *XDB*. The actual Parallel Execution Plan results from the *Parallel Compile Plan* and the selected materialization configuration. Each operator in a Parallel Execution Plan represents a sub-plan of the Parallel Compile Plan that is executed over the horizontally partitioned database (i.e., one node is used to represent the sub-plan over each partition). Figure 3 (left hand side) shows a Parallel Execution Plan for the Parallel Compile Plan in Figure 3. This Parallel Execution Plan has two levels whereas the first level materializes the output of the repartition operator (i.e., intermediate result 6); i.e., the nodes 1.1 to 1.3 of the *Parallel Execution Plan* execute the same sub-plan over different partitions of the database (i.e., 3 partitions in the example), which materialize the intermediate result 6 of the Parallel Compile Plan.

Moreover, a Parallel Execution Plan always executes the same sub-plan in one level l, however, over different partitions. In order to simplify our cost model, we use an Abstract Execution Plan that uses only one operator to represent each level (see right hand side of Figure 3). The idea of an Abstract Execution Plan is that levels are executed sequentially whereas each level materializes its intermediate result. Moreover, we assume that levels are independent from each other (i.e., if one sub-plan in a level fails, it can recover from the materialized result in the level below). In *XDB*, we achieve this independence guarantee by materializing the intermediate results of a sub-plan to a fault-tolerant shared disk (e.g., a shared disk such Amazon EBS when running *XDB* on EC2 instances).

The second input (2) to the cost function is the time required to execute a given sub-plan (which includes its materialization costs). We define the runtime of a level as the maximum runtime of all sub-plans in that level (e.g., the runtime of level $l = 1$ is the maximum runtime of the sub-plans 1.1 to 1.3 in Figure 3). The runtime of an individual sub-plan can be derived by using standard cost models found in databases. It is important to note that we include the materialization costs for each sub-plan into the estimated runtime (e.g., runtime $t_{1.1}$ includes the costs to materialize its intermediate result). Moreover, the network cost for transferring data between Compute Nodes is also included in the consuming sub-plan (e.g., runtime $t_{2.1}$ includes the costs to ship the data over the network).

The last input to the cost function is (3) the MTBF and (4) the MTTR of the Compute Nodes, which can be derived from cluster statistics.

Wasted Time: For the given input, the cost function then enumerates all possible failure scenarios in order to estimate the average wasted time W (i.e., the execution time that is lost due to potential node failures). For a Abstract Execution Plan with L levels, we enumerate L different failure scenarios whereas a individual scenario l represents the case where a node executing a sub-plan in level l fails. For example, Figure 4 shows all possible failure scenarios for the *Abstract Execution Plan* of Figure 3 with two levels: Scenario (1) represents the case where one or more sub-plans in level $l = 1$ fail, and Scenario (2) represents the case where one or more sub-plans in level $l = 2$ fail.

The x-axis in each scenario represents the point in time when the node failure(s) happen during query execution, whereas the y-axis represents the wasted time. Intuitively for Scenario 1, this means the following: if a node in level 1 fails while level 2 is being executed (i.e., a failure happens in the range of $(t_1, t_1 + t_2]$ on the x-axis), we do not waste any time. However, if a node in level 1 fails while level 1 is being executed (i.e., a failure happens somewhere in the range of $[0, t_1]$), we waste the time spend in level 1.

For each scenario l, the average wasted time w_l is thus given by $0.5 \cdot t_l$ whereas t_l is the runtime of level l. In a two-level plan (as in our example), the average wasted time for each three scenarios is thus defined as follows: $w_1 = (0.5 \cdot t_1)$ and $w_2 = (0.5 \cdot t_2)$.

Figure 3 shows that each level consists of one or more sub-plans. The probability that one sub-plan in level l fails is denoted by f_l, and can be derived for a given MTBF as follows: $f_l = (1 - e^{(-t_l/MTBF)})$ where t_l represents the runtime of that level. It is worth noting that probability f_l increases with increasing runtime t_l of that level. Consequently, the probability that one sub-plan in level l succeeds is $s_l = 1 - f_l$.

To this end, the failure scenarios above are not equally likely to happen. For a scenario l, we can calculate its probability p_l as follows. We therefore define $p_l = F_l$, whereas F_l represents the probability that at least one node in level l fails. Moreover, $S_l = 1 - F_l$ is the probability that all nodes in level l succeed, which means that $S_l = (s_l)^n$ where n represents the number of parallel sub-plans in that level (for example, $n = 3$ for both levels in Figure 3).

Finally, based on these findings, we define the average wasted time for a two level plan as $W = \sum_{l=1}^{L} p_l \cdot w_l$, which is $W = p_1 \cdot w_1 + p_2 \cdot w_2$ in the example above.

Estimated Total Runtime: In order to estimate the total runtime T under the presence of node failures for the given input parameters, the idea is to sum up the estimated runtime R of a given query (without a node failure) and the additional runtime resulting from node failures. The additional runtime resulting from node failures is the wasted time W and the mean-time-to-repair multiplied by the number of attempts a needed to achieve a certain success rate for finishing the query (i.e., until the query produces its final result). The runtime R includes the materialization costs for a given materialization configuration.

We first look into the definition of the success rate $S(A \leq a)$ of a query up to attempt a (where $a = 0$ for the first attempt). The success rate of one attempt is $S = S_1 \cdot S_2$ for a two-level plan (remember that S_l is the success rate of a level l) and the failure rate of an attempt is $F = 1 - S$. The success rate up to attempt $a = 0$ is thus defined as $S(A \leq 0) = S$. In general, the success rate up to attempt a (for $a \geq 0$) is: $S(A \leq a) = S + S \cdot F + S \cdot F^2 + ... + S \cdot F^a$. The intuition is that $S(A \leq a)$ defines the cumulative probability that the query finishes in any of the attempts from $A = 0$ to $A = a$.

The formula to calculate $S(A \leq a)$ represents a geometric series. Thus, for $a \to \infty$ the success rate of the query is $S/(1 - F) =$

$S/S = 1$. For a given finite a, the success rate is $S(A \leq a) = S \cdot (1-F^{(a+1)})/(1-F) = (1-F^{(a+1)})$. That way, we can derive the number of attempts a needed to achieve a given success rate S (e.g., of 99%), which is a configuration knob in the cost model.

Based on the given number of attempts, we can now estimate the total runtime T of a query (under node failures) as follows: $T = R + W \cdot F + W \cdot F^2 + W \cdot F^3 + \ldots\ldots + W \cdot F^a + a \cdot MTTR = R + (W + W \cdot F + W \cdot F^2 + \ldots\ldots + W \cdot F^a) - W + a \cdot MTTR$. MTTR is the time to repair a sub-plan: i.e., to redeploy it to another node (in case we replicated partitions) or the time until a failed node needs to be restarted (if we do not replicate partitions). The second term of this formula is again a geometric series. Thus, we can easily calculate T as $T = R + W \cdot (1-F^{(a+1)})/(1-F) - W + a \cdot MTTR$. If we set $F = 0$ and $a = 0$ in this formula (i.e., no node failure happens), then $T = R$ holds.

4. DOOMDB DEMO

In this demo, we showcase our novel cost-based fault-tolerance scheme in *XDB* by presenting a visual computer game called *DoomDB*. *XDB* will be deployed on a Amazon AWS cluster of 10 m1.medium EC2 instances (representing weak commodity machines). The goal of the game is that a user kills database nodes and thus prevents the query from finishing in a given time frame. If *XDB* can not finish the query in a given time, the user wins the game.

When a user starts the game, she can either choose to start a new game or to join an existing game in multi-player mode. When starting a new game, the user can enter the following input parameters:

- **Database:** A pre-partitioned TPC-H database (with 10 partitions per table) of different scaling factors (i.e., $SF = 10$ for short running games and $SF = 100$ for long running games) can be selected.
- **SQL Query:** A SQL query from a subset of the TPC-H benchmark queries can be selected.
- **MTBF:** The mean-time-between-failures of Compute Nodes can be given to influence the materialization configuration.
- **MTTR:** The mean-time-to-repair Compute Nodes can be given.
- **Degree of difficulty ($d \geq 2$):** One of these degrees can be chosen by the player. The higher the degree d is, the more time is given to *XDB* to finish the query (i.e., $d \cdot T$ is given to the player to kill the query where T is the estimated runtime with failures). Thus, for higher degrees the harder it is for players to prevent *XDB* of finishing the query.

After entering the input parameters, the game is started and the user can kill Compute Nodes represented as boxes (see Figure 5). Boxes have guards, which protect the Compute Nodes. Once a Compute Node is killed, its process is stopped. *XDB* then re-deploys the aborted sub-plans of the killed Compute Node and restarts these sub-plans after the Compute Node is repaired (i.e., after the given MTTR). The progress of the query execution is displayed in *DoomDB* while the game is running.

5. RELATED WORK

Two novel approaches, which tackle a similar problem as *XDB* are FTOps [5] and Osprey [9].

FTOps presents a cost-based model for intra-query fault-tolerance without blocking and an optimizer to find the best strategy for an operator (nothing, check-pointing or materializing). Compared to *XDB*, FTOps uses a cost model, which does not define the probability of a node failure based on the runtime of a query. Instead it uses fixed constants for the failure probability.

Osprey is an approach, which is only working for analytical queries over a star schema. It first splits the analytical queries into

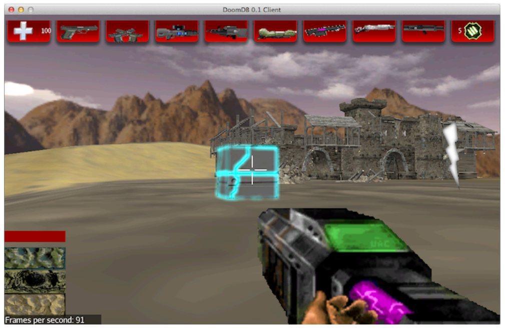


Figure 5: DoomDB Gaming Screen

sub-queries over partitions and then executes a final merge over all intermediate results of sub-queries. If a sub-query fails, it is restarted on a different replica. *XDB* extends the ideas presented in Osprey to arbitrary schemata and queries. Moreover, *XDB* also presents a cost model to find the best deployment under different failure rates.

6. CONCLUSIONS AND OUTLOOK

In this demo paper we presented our parallel database system called *XDB*, which is implemented using a middleware approach over an existing database system. To showcase the cost-based fault-tolerance scheme in *XDB*, we implemented an ego-shooter computer game called *DoomDB*. The goal of the game is to kill an analytical SQL query running in a fault-tolerant environment on a cluster of database nodes. One important avenue of future work is to extend our current cost model for bushy plans as well as for arbitrary graph-based plans (instead of left-deep plans only).

7. ACKNOWLEDGMENTS

First, we would like to thank the team of students for implementing the DoomDB game: Kai Boernert, Daniel Ingenhaag, Lucas La Salle, Stefano Enzo Leccese of University of Mannheim. Moreover, we also would like to thank Amazon for providing an Amazon AWS in Education Research Grant for the student team.

8. REFERENCES

[1] C. Binnig, N. May, and T. Mindnich. SQLScript: Efficiently Analyzing Big Enterprise Data in SAP HANA. In *BTW*, pages 363–382, 2013.

[2] C. Binnig, A. Salama, E. Zamanian, A. C. Müller, S. Listing, and H. Kornmayer. XDB - A novel Database Architecture for Data Analytics as a Service. Technical report, University of Mannheim, 2013. `http://pi1.informatik.uni-mannheim.de/filepool/publications/XDB/xdb_techreport.pdf`.

[3] Cloudera. Impala. `http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html`.

[4] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[5] P. Upadhyaya, Y. Kwon, and M. Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *SIGMOD Conference*, pages 241–252, 2011.

[6] F. M. Waas. Beyond Conventional Data Warehousing - Massively Parallel Data Processing with Greenplum Database. In *BIRTE (Informal Proceedings)*, 2008.

[7] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[8] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD Conference*, pages 13–24, 2013.

[9] C. Yang, C. Yen, C. Tan, and S. Madden. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *ICDE*, pages 657–668, 2010.