Fast DDL

Teju Nareddy
tejunareddy@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia

Grace Harper
gharper31@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia

Sai Gundlapalli
sgundlapalli@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia

KEYWORDS

query evolution. data definition language

1 INTRODUCTION

Schema evolution studies the issue of moving a database from one version of its schema to a new updated schema. The database community has been interested in how to change schemas in a structured, scalable, and safe manner for many years and the challenge to do this boils down to a few reasons. Schema changes (DDL operations) vary in complexity, ranging from simply adding a nullable column to decomposing a single table into two tables with subsets of columns from the original table. Some of these changes require large amounts of data to be moved from table to table causing large strains on system resources and increasing the likelihood of errors. Online schema operations are modifications of the table structure without locking down the entire table(s) for the duration of the DDL operation [Noach 2009]. Historically, database administrators have to perform these tasks offline that includes a lot of manual work and custom scripting as the traditional SQL interface causes downtime to databases that certain services can simply not afford. The benefit of this approach is its safety and reliability since no data change other than those related to the schema change can occur when the service is offline.

2 BACKGROUND

This section introduces some background on different aspects of database systems we believe is helpful in understanding why DDL operations are slow and what we need to focus on to attain faster DDL operations.

2.1 Schema Update Strategies

We will take a look at the work necessary to complete schema changes and why it may be difficult to support online schema evolution. Different database systems have different locking mechanisms which will change the performance of different schema changes.

In-place update strategy

An in-place update strategy will have a background thread modifying the table while foreground threads still query the table for data. This strategy is supported natively by SQL DML statements and is a commonly used approach today [Zhu 2017].

Some DDL operations will only change the metadata of the table and not the data in the table itself. These operations can be done very quickly and efficient support for this operation is supported in database systems. It is more efficient because the schema update does not require locking the entire table but a short-lived latch on the metadata structure.

DDL operations that also require changing the data in the table will need to acquire the locks to resources they are modifying which conflicts with other user provided queries reducing the performance.

Copy-based update strategy

An alternate strategy would be to leave the original table unchanged by the DDL operation but create a copy of the table with the desired schema structure and copy the old table into the new table. This approach avoid contention over exclusively locking resources (rows or indexes) but introduces background work which impacts CPU usage. Furthermore, by creating a copy of the original table, there needs to be a synchronization mechanism in place to ensure correctness of the database system.

3 MOTIVATION

A database schema describes a structure of the database in a formal language. As the needs for the real world application changes, the underlying database schema will change in response. These DDL operations are cumbersome and result in potential downtime to production systems that certain organizations simply cannot afford [Zhu 2017]. There is much value that can be gained from an operations perspective to enhance current continuous integration and deployment pipelines [de Jong and van Deursen [n. d.]]. Furthermore, modern web application frameworks deeply integrate object-relational mappings (ORM) tools allowing application developers the ability to randomly modify the schema [Verbitski et al. 2017]. For example, TripAdvisor regularly has half a dozen schema changes per week [Adrian 2015]. This problem is real and affects many large companies operating today, some seeking solutions for zero-downtime and some accepting the maintenance window [jbaviat 2018]. We will attempt to design a faster DDL implementation to allow for online schema upgrades.

4 RELATED WORK

We are aware of a small number of pre-existing implementations of Fast DDL.

Facebooks Open Schema Change for MySQL is a separate tool that can coexist on the database server and manage schema migrations at the table level. Its main goal is to limit the number of exclusive locks set on the original table, allowing other transactions

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to make forward progress during a schema change. OSC accomplishes this via multiple phases. It first copies the original table asynchronously without blocking other transactions. It then replays all changes made to the original table during the copy phase. Finally, a database engineer specifies exactly when the database should cut-over to the new copy [Callaghan 2010].

QuantumDB is a tool similar to OSC. It also uses the copy, build, replay, and cut-over approach [quantumdb.io/ 2018].

Amazon Aurora ships Fast DDL as a first-class feature of its fully-managed AWS RDS service. Unlike OSC and QuantumDB, this feature is directly built into the DBMS and executes incremental schema changes to the underlying pages during queries instead of automatically updating the entire table at once. This allows DDL statements to run almost instantaneously. However, it is unclear whether incrementally updating tuples would significantly slow down subsequent queries, as the documentation for this feature does not expose such metrics. Additionally, Aurora only offers Fast DDL on a few specific commands, such as adding a column of nullable values to a pre-existing table. [Gupta 2017]

Zhu's proxy-based schema evolution system, Ratchet [Zhu 2017], implements fast DDL by breaking down the schema changes into small and manageable components. However, Ratchet does not offer instantaneous DDL like Aurora [Zhu 2017].

Another online schema tool is Percona's pt-online-schema-change, which is part of the Percona toolkit . When a schema update is enacted, the tool first creates a new table with the desired schema. A trigger is added to the old table to allow for changes on the original table propagate to the new table. Then a background task then copies over all the data from the original table to the new table. An atomic instruction is executed to the rename the new table to the original table [Ksiazek 2016]. Although an improvement, overhead is still incurred from inserting data into a table and triggers.

5 PROBLEM STATEMENT

Given the need for better DDL across industry, we look to design a faster DDL implementation that allows for near instantaneous schema modification while reducing the runtime impact on subsequent queries. Specifically, we aim to create a DDL implementation that allows for a broader range of near-instantaneous DDL commands. Additionally, we aim to optimize the performance of queries directly after the near-instantaneous DDL operation. Previous metrics published from the industry show that every 100ms of latency costs Amazon 1% of profit [Torkington 2008]. Optimizing for this case could be mission-critical in the industry.

Our proposed solution is a combination of solutions generated by previous work. In order to achieve instantaneous modifications for a wider range of DDL command we look to develop on top of Aurora's previous work [Gupta 2017], looking at how each DDL modification can be stratified across queries and on Zhu's Ratchet [Zhu 2017] to see how complex DDL updates can be broken down into smaller update components. We hope that by combining these two subdivision approaches, we can achieve fast DDL for a wider variety of commands. Namely, we look to create fast DDL for adding a column with data to a table and for changing the datatype of the preexisting column. And that this double-stratified solution will provide a generic-framework applicable to many other types of DDL commands beyond the scope of this project.

6 ASSUMPTIONS

We make a number of assumptions for this work so that we can complete the project in an efficient manner:

- We aim to develop a Fast DDL implementation for databases using the NSM storage model. Changing table schemas in a DSM storage model is trivial, so we will not focus on that.
- We will only support Fast DDL on a subset of difficult DDL operations. For example, attribute deletion is trivial, so we will not focus on that.
- We simplify the problem space significantly by not supporting database version changes or durability through crashes.
- Considering multiple DDL commands on the same set of data is also beyond the scope of this project. We will only support a single DDL statement at a time in our prototype.

7 APPROACH

7.1 Aurora Benchmarking

Initially, we planed on benchmarking the impact of Amazon Aurora's Fast DDL feature on queries run directly after the DDL operation. This is an essential first step because the metrics generated will guide our direction on the remainder of the project. If we find that response times of queries after DDL operations are substantially larger than those of standard queries (which we expect to see), we will prioritize reducing this impact in our fast DDL implementation. Otherwise, we will can focus on achieving instantaneous DDL without optimizing for subsequent queries.

To run this benchmark, we planned to generate our own workload instead of using a pre-existing one. Most pre-existing workloads focus on DML operations, not DDL operations. A workload such as the Schema Evolution Benchmark does support DDL statements, but would not expose the worst-case response times on subsequent queries in Amazon Aurora [Yel [n. d.]]. We require our own custom benchmark where we can tune table schemas, attribute sizes, and amount of data. We will then measure the runtime of queries executed directly after an expensive DDL operation. We will then compare this to the runtime of queries executed much later, when all the underlying pages have been shifted to the new schema. We can then determine if the DDL operation had a significant impact on the runtime of subsequent queries.

7.2 Custom Test Bed

After meeting with our research advisor, we decided that benchmarking Aurora would offer very little information gain because it is a "black box". The source code for Amazon Aurora is proprietary, making it difficult to validate our results. In the case that our benchmarks generate results we do not expect, we cannot drill down to understand the root cause. Additionally, we will need a test bed to test any future custom DDL implementations.

Instead of benchmarking Amazon Aurora directly, we created a test bed with mock implementations of a standard MySQL table and an Amazon Aurora Table. We then ran the benchmarks described above on these mock table implementations. The results of these benchmarks are further described in 10.

This test bed would be more realistic if implemented on a fully-functional BadgerDB, but we can also implement it using common STL data structures in C++. For example, we assume all tables only hold integer attributes. Additionally, we only support adding columns to a row-oriented table. Finally, we assume all data is stored in memory.

7.3 Custom DDL Implementation

Once we complete the benchmarks and decide the priority of subsequent query impact, we will start prototyping a Fast DDL implementation. This will be prototyped directly in the test bed, allowing us to run the same benchmarks to compare and contrast it with preexisting implementations.

The exact steps of prototyping are highly dependent on further research we complete and the results of the aforementioned benchmark. Therefore, we do not present further details on it.

8 FEATURES TO SUPPORT

As previously mentioned, we plan to focus on two main factors: Near-instantaneous DDL and low impact on subsequent queries. Additionally, we aim to support and optimize for the following scenarios in our implementation:

- Schema evolution should not overload CPU and I/O, as that results in starvation in processes.
- Transactions should be able to add new data to a table while a schema change is in progress. In other words, exclusive locking should be kept to a minimum.
- The database must be online and sufficiently available to ensure a certain level of quality for the applications using the database.
- Long-running background processes should be minimized, as they could significantly slow down recovery.

9 IMPLEMENTATION VALIDATION

Currently, we aim to use the MediaWiki dataset in order to evauluate the effectiveness of our system in handling schema changes [Yel [n. d.]]. We will need to translate the mysql initialization script required to be compatible with our prototype and populate our database with data.

10 PERFORMANCE EVALUATION

We would like to evaluate our projects ability to handle scheme upgrade requests while the system is online. Specifically, we will need to measure throughput and latency of DML operations and how they are impacted by our implementation of Fast DDL operations.

10.1 Aurora Long Tail Latency

To understand the trade-offs of Amazon Aurora's Fast DDL feature, we executed unit tests to measure average/max/min/total latency of DML operations and individual tuple query access times across multiple runs. Specifically, we ran our benchmarks on three types of tables:

Table 1: Operation Time (ms) by Table Type

	Table Type	
Operation (ms)	Naive Random Memory	Aurora
Add Tuples	3.82	4.29
Full Scan (pre-DDL)	3.15	3.63
DDL (add columns)	13.67	1.24e-4
Full Scan (initial)	6.61	21.03
Full Scan (avg)	6.23	6.40

Table 2: Aurora Post-DDL Individual Tuple Query Time

Operation	ms
Total	21.03
Max	2.99
Average	0.04
Min	7.5e-4

(1) **Naive Contiguous Memory Table**: A naive MySQL table, where a DDL operation copies the entire table to a new table. All tuples in this table are stored contiguously in memory.
(2) **Naive Random Memory Table**: A naive MySQL table, where a DDL operation copies the entire table to a new table. All tuples within tuple groups (pages) are stored contiguously in memory, but tuple groups themselves are located on random sections of the heap. This better represents an on-disk database, so we used this table as a baseline for our Fast DDL benchmark.
(3) **Aurora Table**: A mock implementation of the Amazon Aurora's Fast DDL feature, where tuple groups are lazily copied on tuple access. This was the main table under test and was compared to the benchmarks from the naive table's baseline.

We ran the following operations (in order) on each table, measuring latency as described above:

(1) Create a table of 1024 integer attributes
(2) Add $8 * 64$ tuples to fill the table
(3) Full scan to ensure all $8 * 64$ tuples are persisted in memory
(4) DDL: Increase the table size to store an extra 1024 integer columns
(5) Full scan (multiple times) to measure latency after DDL

Table 1 displays the latency of each operation described above the tables.

As expected, the DDL time for the Naive Table blocks longer than the rest of the operations. Once the DDL is complete, all future Full Scans take about the same amount of time. However, this is not the case for the Aurora implementation. In Aurora, the DDL operation executes in microseconds because it only changes metadata. However, the subsequent full scan directly after the DDL statement takes much longer to execute than future full scans. This occurs because Aurora copies over all tuple groups in the subsequent full scan.

We drilled down into the 21.03 ms Full Scan in the Aurora table. Table 2 displays the latency to access individual tuples directly after the DDL operation.

We noticed that a small number of tuples took 75 times longer to access than most other tuples. This is an inherent trade-off of Amazon Aurora's Fast DDL implementation, and is similar to the *long tail latency problem.*

10.2 Future Benchmarks

Our future experiments will consist of a test application that will insert records into a single table and starts a demo application with two threads manipulating the data; one thread will be executing normal DML operations updating data in the database and the other thread will be doing DDL operations and updating the schema. The DDL statements we will test have not been decided at the current stage.

We will execute the schema updates using the normal, supported approach (normally ALTER TABLE statement) and then using our implementation of fast DDL. During this process, we keep track of the duration of the queries issued by the application and how long they take to complete.

We will need a fully-functioning multi-versioned concurrency control implementation in our test bed to run such queries.

11 PROGRESS AND TIMELINE

This section will cover the goals our team has set out for the project; it will include 75%, 100%, and 125% goals corresponding to a C, B, and A letter grade respectively. We will also cover the progress we have made on these goals up until the writing of this report. Thus far we have met our 75% goal of a mock implementation of Aurora and successfully benchmarked it against a naive DDL implementation. For our 100% goal we aim to incorporate MVCC and allow for concurrent access to tuples. We plan on discussing the 125% goal with the professor in our upcoming meeting this week.

Initially, the group planned to benchmark Amazon Aurora in isolation to see the impact on subsequent queries but because we cannot compare these metrics to other implementations this was nixed. We instead attempt to implement Aurora on our custom database system implementation to benchmark against other DDL optimizations.

Our current implementation for the database system is covered in section 10.1. This currently is single-threaded and any DDL changes will block any user transactions.

Moving forward, the next optimization we want to test is a MVCC approach to DDL changes. To support concurrent access on a row level, we will store different versions of the tuple as a DDL update is modifying it. This can be extended to tuple groups as well. In attempt to save memory and improve performance, we will try to delay materialization of the schema change as late as possible. In this sense, there are some parallels between Amazon Aurora's approach.

Once we implement concurrent access between DML and DDL operations, we will update our testbed to reflect this new functionality.

Furthermore, time permitting, it would be interesting to apply some of the optimizations found in current work to a functioning dummy database system such as BadgerDB or SimpleDB. Converting some of the in-place schema updates to copy-over and leveraging triggers will be interested to compare to the current implementation.

We discuss the timeline moving forward here.

- **April 1-7:** Discuss architecture and class structure of MVCC, begin implementation, unit test coverage for MVCC implementation
- **April 8-14:** Code review, finalize MVCC implementation and test coverage, incorporate continuous integration for unit tests and integration tests
- **April 15-21:** Gather various metrics i.e. latency, cpu usage, and benchmark in CoC sandbox environment. Optimize more complex schema modification
- **April 22-24:** Continue previous week's work, prepare final report, prepare presentation and demo
- **April 25:** Project Presentation

REFERENCES

[n. d.]. Schema Evolution Benchmark. http://yellowstone.cs.ucla.edu/schema-evolution/index.php/Schema_Evolution_Benchmark

Merv Adrian. 2015. Retrieved March 30, 2019 from https://twitter.com/merv/status/667048388958011392

Mark Callaghan. 2010. Online Schema Change for MySQL. Retrieved March 7, 2019 from https://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932/

Michael de Jong and Arie van Deursen. [n. d.]. Continuous Deployment and Schema Evolution in SQL Databases. *IEEE Explore* ([n. d.]).

Anurag Gupta. 2017. Amazon Aurora Under the Hood: Fast DDL. Retrieved March 7, 2019 from https://aws.amazon.com/blogs/database/amazon-aurora-under-the-hood-fast-ddl/

jbaviat. 2018. Managing database schema changes without downtime. Retrieved March 30, 2019 from https://news.ycombinator.com/item?id=16665447

Krzysztof Ksiazek. 2016. Online schema change for MySQL MariaDB - comparing GitHubâĂŹs gh-ost vs pt-online-schema-change. Retrieved March 30, 2019 from https://severalnines.com/blog/online-schema-change-mysql-mariadb-comparing-github-s-gh-ost-vs-pt-online-schema-change

Shlomi Noach. 2009. Online ALTER TABLE now available in operark kit. Retrieved March 30, 2019 from http://code.openark.org/blog/mysql/online-alter-table-now-available-in-openark-kit

quantumdb.io/. 2018. QuantumDB. github.com/quantumdb/quantumdb

Nat Torkington. 2008. Radar Theme: Web Ops. http://radar.oreilly.com/2008/08/radar-theme-web-ops.html

Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17).* ACM, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

Yu Zhu. 2017. *Towards Automated Online Schema Evolution.* Ph.D. Dissertation. University of California at Berkley. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-218.pdf