Improving OCC by Transaction Batching and Reordering

Bailu Ding\textsuperscript{1}, Lucja Kot\textsuperscript{2}, Johannes Gehrke\textsuperscript{3}

\textsuperscript{1}Microsoft Research, \textsuperscript{2}Gramma Tech, Inc, \textsuperscript{3}Microsoft Corporation
DMX Group Overview

- Data management, exploration and mining
- Flexible resource allocation mechanisms and policies for cloud database
  - Contact: Vivek Narassaya
- Self-service data preparation
  - Contact: Yeye He
- Approximate query processing
  - Contact: Surajit
- Actor-Oriented Databases (Orleans, link)
  - Contact: Phil Bernstein
- Automated physical design in the cloud
  - Contact: Sudipto Das, Bailu Ding
- And many more!
Automated Physical Design in the Cloud

- A continuous indexing framework to automatically select and build indexes to reduce query execution time
  - Closed-loop solution: success measured in terms of real execution time instead of query optimizer costs
  - A hands-free solution: remove human intervention from the critical path of the loop

- More than index recommendation: workload extraction, index implementation, validation, and monitoring
  - Automatically Indexing Millions of Databases in Microsoft Azure SQL Database, Sudipto Das, Miroslav Grbic, Igor Ilic, and Et al., SIGMOD 2019

- Improve plan quality with reduced execution cost at low risk with multiple executed plans of the same query
  - Plan Stitch: Harnessing the Best of Many Plans, Bailu Ding, Sudipto Das, Wentao Wu, and Et al., VLDB 2018
Improving OCC Through Transaction Batching and Operation Reordering
Optimistic Concurrency Control

- Read phase
- Validation phase
- Write phase
Optimistic Concurrency Control

- Read phase
- Validation phase
- Write phase

**T1**
- Read (X) -> 10
- Read (Y) -> 20
- Write (X) -> 30

**T2**
- Read (X) -> 10
- Read (Z) -> 50
- Write (Z) -> 40

Read Phase
Optimistic Concurrency Control

- Read phase
- Validation phase
- Write phase
Optimistic Concurrency Control

- Read phase
- Validation phase
- Write phase

Read Phase

T1
- Read (X) -> 10
- Read (Y) -> 20
- Write (X) -> 30

T2
- Read (X) -> 10
- Read (Z) -> 50
- Write (Z) -> 40

Serialize T1 Before T2

X 10
Y 20
Z 50

Commit

T1

T2

30
20
50
Optimistic Concurrency Control

- Read phase
- Validation phase
- Write phase
Is T2 Destined to Abort, Really?

T1
- Read (X) -> 10
- Read (Y) -> 20
- Write (X) -> 30

T2
- Read (X) -> 10
- Read (Z) -> 50
- Write (Z) -> 40

Serialize T1 Before T2

X: 10
Y: 20
Z: 50

T1
- Commit

T2
- Stale read
- Abort

30
20
50
Is T2 Destined to Abort, Really?

**T1**
- Read (X) -> 10
- Read (Y) -> 20
- Write (X) -> 30

**T2**
- Read (X) -> 10
- Read (Z) -> 50
- Write (Z) -> 40

**Serialize T1 Before T2**
- X: 10
- Y: 20
- Z: 50

**T1**
- Commit: 30
  - X: 20
  - Y: 50

**Stale read**

**Abort**

**Serialize T2 Before T1**
- X: 10
- Y: 20
- Z: 50

**T2**
- Commit: 10
  - X: 10
  - Y: 20

**T1**
- Commit: 30
  - X: 40
  - Y: 40
Is T2 Destined to Abort, Really?

- Conflicting concurrent transactions can potentially all commit with an alternative serialization order.
Transaction Reordering in OCC

- Optimistic concurrency control finalizes the serialization order *after* transaction execution
- Incorporate reordering throughout the life of a transaction
- Batch transactions explicitly to open doors for reordering
  - Limit the scope of reordering with a batch
A Life of a Transaction

Client

Transaction Processing

Transaction Coordinators

Read / write request
Batch and reorder R/W operations
Storage

Validation request queue
Batch and reorder transactions
Validation
A (New) Life of a (Batched and Reordered) Transaction

Client

Transaction Processing

Transaction Coordinators

Read / write request

Validation request queue

Batch and reorder R/W operations

Batch and reorder transactions

Storage

Validation
Transaction and Operation Reordering

- Reordering transactions at clients
  - Prior work on static transaction scheduling

- Reordering operations at storage
  - Optimal strategy: Prioritize writes before reads to avoid stale reads

- Reordering transactions at validation
  - How to create a serialization order with the least number of aborts from a batch of transactions?
Transaction Reordering at Validation

- Given a batch of transactions $B$, construct subset $B' \subseteq B$, such that $B'$ is serializable and $|B'|$ is maximal among all $B' \subseteq B$
  - The number of aborts is minimized within the batch if $|B'|$ is maximal
- How to decide if $B'$ is serializable and how to construct a serialization order?
Constructing a serialization order

- Dependency graph: the conflicts of transactions
  - Example: A transaction T1 cannot be serialized after a transaction T2 if T2 updates an item it reads. We have T2 -> T1.
Constructing the Maximal $B'$

- Given $B$ and a dependency graph $G(B)$, find the maximal $B' \subseteq B$ such that $G(B')$ is acyclic.
- Find the minimal $V \subseteq B$ such that $G(B \setminus V)$ is acyclic.
  - $B' = B \setminus V$
- Feedback vertex set!
- But it is NP hard ...
- Greedy algorithms
  - Strongly connected component (SCC) based
  - Sort based: more efficient but less accurate
Policies for Alternative Performance Goals

- Minimize $V$: minimize the number of aborts
- Alternative goals
  - Minimize tail latency
  - Minimize the number of restarts
  - Maximize monetary value
- Weighted feedback vertex set
  - SCC and sort based greedy algorithms
Evaluation: Write-Intensive Skewed YCSB

- Up to 2.2x improvement in throughput
Evaluation: Write-Intensive Skewed YCSB

- Up to 4x reduction in percentile latency

![Graph showing percentile latency for Reorder, Silo, Cicada, and TicToc]

- 90%
- 95%
- 99%

Latency / MS
Optimize for Tail Latency

- Up to 6.3x reduction in tail latency

![Graph showing latency reduction]
Conclusion

- Explicit batching opens doors for transaction and operation reordering in optimist concurrency control.
- Batching and reordering transactions improve throughput and reduce tail latency.
- Weighted reordering policies enable optimization for alternative performance goals such as tail latency and monetary value.
Towards a Learning Optimizer for Shared Clouds*

Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, Sriram Rao

February 8, 2019

Rise of Big Data Systems

Hive
Spark
Flink
Calcite
BigQuery
Big SQL
HDInsight
SCOPE
Etc.

Declarative query interface
Cost-based query optimizer (CBO)
Rise of Big Data Systems

- Hive
- Spark
- Flink
- Calcite
- BigQuery
- Big SQL
- HDInsight
- SCOPE
- Etc.

Declarative query interface
Cost-based query optimizer (CBO)

```sql
SELECT Customer.cname, Item.iname
FROM Customer
INNER JOIN Order
ON Customer.cid == Order.cid
INNER JOIN Item
ON Item. iid == Order. iid
WHERE Item.price > 100
AND Customer.cage < 18;
```
Rise of Big Data Systems

Hive
Spark
Flink
Calcite
BigQuery
Big SQL
HDInsight
SCOPE
Etc.

Declarative query interface
Cost-based query optimizer (CBO)

```
SELECT Customer.cname, Item.iname
FROM Customer
INNER JOIN Order
ON Customer.cid == Order.cid
INNER JOIN Item
ON Item.iid == Order.iid
WHERE Item.iprice > 100
AND Customer.cage < 18;
```

Good plan => Good performance
Problem: CBO can make mistakes esp. Cardinality Estimation
Rise of Big Data Systems

Hive
Spark
Flink
Calcite
BigQuery
Big SQL
HDInsight
SCOPE
Etc.

The root of all evil, the Achilles Heel of query optimization, is the estimation of the size of intermediate results, known as cardinalities. – [Guy Lohman, SIGMOD Blog 2014]
Rise of Big Data Systems

BIG DATA SYSTEM

Hive
Spark
Flink
Calcite
BigQuery
Big SQL
HDInsight
SCOPE
Etc.

TUNING!
Collecting Statistics
Providing Query Hints
Database Administration
Rise of the Clouds

BIG
DATA
SYSTEM

Hive
Spark
Flink
Calcite
BigQuery
Big SQL
HDInsight
SCOPE
Etc.

Collecting Statistics
Providing Query Hints
Database Administration
Rise of the Clouds

BIG DATA SYSTEM

Hive
Spark
Flink
Calcite
BigQuery
Big SQL
HDInsight
SCOPE
Etc.

MANAGED SERVERLESS

Collecting Statistics
Providing Query Hints
Database Administration
Rise of the Clouds

BIG DATA SYSTEM

Hive
Spark
Flink
Calcite
BigQuery
Big SQL
HDInsight
SCOPE
Etc.

MANAGED SERVERLESS

No Admin
No Expertise
No Control

Collecting Statistics
Providing Query Hints
Database Administration
Rise of the Clouds

BIG DATA SYSTEM

Hive
Spark
Flink
Calcite
BigQuery
Big SQL
HDInsight
SCOPE
Etc.

SELF TUNING!
Hope: Shared Cloud Infrastructures

Massive volumes of query logs

Shared data processing
Hope: Shared Cloud Infrastructures

- Big Data System
  - Massive volumes of query logs
  - Centrally visible query workload

- Shared data processing
Cosmos: shared cloud infra at Microsoft

• SCOPE Workloads:
  • Batch processing in a job service
  • 100Ks jobs; 1000s users; EBs data; 100Ks nodes

• Cardinality estimation in SCOPE:
  • 1 day’s log from Asimov
Cosmos: shared cloud infra at Microsoft

- SCOPE Workloads:
  - Batch processing in a job service
  - 100Ks jobs; 1000s users; EBs data; 100Ks nodes

- Cardinality estimation in SCOPE:
  - 1 day’s log from Asimov
  - Lots of constants for best effort estimation
  - Big data, unstructured Data, custom code
Cosmos: shared cloud infra at Microsoft

- SCOPE Workloads:
  - Batch processing in a job service
  - 100Ks jobs; 1000s users; EBs data; 100Ks nodes
- Cardinality estimation in SCOPE:
  - 1 day’s log from Asimov
  - Lots of constants for best effort estimation
  - Big data, unstructured Data, custom code
- Workload patterns
  - Recurring jobs
  - Shared query subgraphs

![Graph showing fraction of subgraphs over estimate/actual cardinality ratio]
Cosmos: shared cloud infra at Microsoft

- SCOPE Workloads:
  - Batch processing in a job service
  - 100Ks jobs; 1000s users; EBs data; 100Ks nodes

- Cardinality estimation in SCOPE:
  - 1 day’s log from Asimov
  - Lots of constants for best effort estimation
  - Big data, unstructured Data, custom code

- Workload patterns
  - Recurring jobs
  - Shared query subgraphs
Cosmos: shared cloud infra at Microsoft

- **SCOPE Workloads:**
  - Batch processing in a job service
  - 100Ks jobs; 1000s users; EBs data; 100Ks nodes

- **Cardinality estimation in SCOPE:**
  - 1 day’s log from Asimov
  - Lots of constants for best effort estimation
  - Big data, unstructured Data, custom code

- **Workload patterns**
  - Recurring jobs
  - Shared query subgraphs

- **Can we learn cardinality models?**
Learning Cardinality Model

- **Strict**: cache previously seen values
  - Low coverage
  - Online feedback
- **General**: learning a single model
  - Hard to featurize
  - Hard to train
  - Prediction latency
  - Low accuracy
- **Template**: learning a model per subgraph template

$=>$ *No one-size-fits-all*
Learned Cardinality Models

- **Subgraph Template:**
  - Same logical subexpression
  - Different physical implementation
  - Different parameters and inputs

- **Feature Selection**

- **Model Selection**
  - Generalized liner models due to their interpretability
  - More complex models, such as multi-layer perceptron harder to train

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JobName</td>
<td>Name of the job containing the subgraph</td>
</tr>
<tr>
<td>NormJobName</td>
<td>Normalize job name</td>
</tr>
<tr>
<td>InputCardinality</td>
<td>Total cardinality of all inputs to the subgraph</td>
</tr>
<tr>
<td>( \text{Pow}(\text{InputCardinality}, 2) )</td>
<td>Square of InputCardinality</td>
</tr>
<tr>
<td>( \text{Sqrt}(\text{InputCardinality}) )</td>
<td>Square root of InputCardinality</td>
</tr>
<tr>
<td>( \text{Log}(\text{InputCardinality}) )</td>
<td>Log of InputCardinality</td>
</tr>
<tr>
<td>AvgRowLength</td>
<td>Average output row length</td>
</tr>
<tr>
<td>InputDataset</td>
<td>Name of all input datasets to the subgraph</td>
</tr>
<tr>
<td>Parameters</td>
<td>One or more parameters in the subgraph</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>Percentage Error</th>
<th>Pearson Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Optimizer</td>
<td>2198554</td>
<td>0.41</td>
</tr>
<tr>
<td>Adjustment Factor (LEO)</td>
<td>1477881</td>
<td>0.38</td>
</tr>
<tr>
<td>Linear Regression</td>
<td>11552</td>
<td>0.99</td>
</tr>
<tr>
<td>Neural Network</td>
<td>9275</td>
<td>0.96</td>
</tr>
<tr>
<td>Poisson Regression</td>
<td>698</td>
<td>0.98</td>
</tr>
</tbody>
</table>
Accuracy: 10-fold cross validation

![Graph showing accuracy for different methods. The x-axis represents the estimated/actual cardinality ratio, while the y-axis represents the fraction of subgraph instances. The graph compares Neural Network, Linear Regression, and Poisson Regression.]
Accuracy: 10-fold cross validation

![Graph showing accuracy comparison between Neural Network, Linear Regression, and Poisson Regression. Graph includes a table showing the 75th and 90th percentile errors for Default SCOPE and Poisson Regression.]

Note: Neural network overfits due to small observation and feature space per model.
Applicability: %tage subgraphs having models

Varying Training Window

![Applicability Chart](image)

- **Jobs**
- **Subgraphs**

<table>
<thead>
<tr>
<th>Train Duration</th>
<th>Jobs</th>
<th>Subgraphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-day</td>
<td>60%</td>
<td>40%</td>
</tr>
<tr>
<td>2-day</td>
<td>65%</td>
<td>35%</td>
</tr>
<tr>
<td>4-day</td>
<td>70%</td>
<td>30%</td>
</tr>
<tr>
<td>1-week</td>
<td>75%</td>
<td>25%</td>
</tr>
<tr>
<td>2-week</td>
<td>80%</td>
<td>20%</td>
</tr>
<tr>
<td>1-month</td>
<td>85%</td>
<td>15%</td>
</tr>
</tbody>
</table>
Applicability: %tage subgraphs having models

Varying Training Window

Sliding Test Window

<table>
<thead>
<tr>
<th>Train Duration</th>
<th>Applicability (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-day</td>
<td>Jobs</td>
</tr>
<tr>
<td>2-day</td>
<td></td>
</tr>
<tr>
<td>4-day</td>
<td></td>
</tr>
<tr>
<td>1-week</td>
<td></td>
</tr>
<tr>
<td>2-week</td>
<td></td>
</tr>
<tr>
<td>1-month</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Slide Duration</th>
<th>Applicability (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-day</td>
<td>Jobs</td>
</tr>
<tr>
<td>1-week</td>
<td></td>
</tr>
<tr>
<td>1-month</td>
<td></td>
</tr>
</tbody>
</table>
End-to-end Feedback Loop

- Model Lookup & Prediction
  - Compiler
  - Optimizer
  - Scheduler
  - Runtime

Query
Cardinality Models

Model Server
Parallel Trainer
Workload Analyzer

Compiled query DAGs
Optimized plans & estimated statistics
Execution graphs & resources
Actual runtime statistics

Result
End-to-end Feedback Loop

Trained offline over new batches of data
Large number of smaller, **highly accurate** models
End-to-end Feedback Loop

- Model Lookup & Prediction
  - Compiler
  - Optimizer
  - Scheduler
  - Runtime

- Annotation hints to the query optimizer
- Cardinality Models

Model Server
- Parallel Trainer
- Workload Analyzer

- Compiled query DAGs
- Optimized plans & estimated statistics
- Execution graphs & resources
- Actual runtime statistics

Trained offline over new batches of data
Large number of smaller, highly accurate models
End-to-end Feedback Loop

Easy to featurize with low overhead
Accurate and easy to understand

Model Lookup & Prediction

Compiler → Optimizer → Scheduler → Runtime → Result

Query

Annotation hints to the query optimizer
Cardinality Models

Model Server
Parallel Trainer
Workload Analyzer

Compiled query DAGs
Optimized plans & estimated statistics
Execution graphs & resources
Actual runtime statistics

Trained offline over new batches of data
Large number of smaller, highly accurate models
Performance

- Subset of hourly jobs from Asimov
- These queries process unstructured data, use SPJA operators, and a UDO
- Re-ran the queries over same production data, but with redirected output
Performance

- Subset of hourly jobs from Asimov
- These queries process unstructured data, use SPJA operators, and a UDO
- Re-ran the queries over same production data, but with redirected output
Performance

- Subset of hourly jobs from Asimov
- These queries process unstructured data, use SPJA operators, and a UDO
- Re-ran the queries over same production data, but with redirected output
Avoiding Learning Bias

• Learning only what is seen
• Exploratory join ordering
  • Actively try different join orders
  • Pruning: discard plans with subexpressions that are more expensive than at least one other plan
  • Maximize new observations when comparing plans
• Execution strategies
  • Static workload tuning
  • Using sample data
  • Leveraging recurring/overlapping jobs
Takeaways

- Big data systems increasingly use cost-based optimization
- Users cannot tune these systems in managed/serverless services
- Hard to achieve a one-size-fits-all query optimizer
- Instance optimized systems are more feasible
- Very promising results from SCOPE workloads:
  - Could achieve very high accuracy
  - Reasonably large applicability, could further apply exploration
  - Performance gains, most significant being less resource consumption
- Learned cardinality models a step towards self-learning optimizers
Machine Learning in Google BigQuery

Amir Hormati (hormati@google.com)
Agenda

BigQuery
Why BigQuery ML?
Syntax
Iterative Gradient Descent
Closed Form Solution
Questions
BigQuery

- Google’s cloud-based SQL datawarehouse-as-a-service for analytics:

  - Enterprise data warehouse for analytics
  - Convenience of standard SQL
  - Fully managed and serverless
  - Petabyte-scale storage and queries
  - Encrypted, durable and highly available
  - Real-time analytics on streaming data

https://cloud.google.com/bigquery
BigQuery ML

- SQL analysts use databases to extract insights from their data.

```sql
SELECT AVG(income) FROM census_data GROUP BY state;
SELECT cid, COUNT(*) FROM orders GROUP BY cid ORDER BY COUNT(*) DESC
```

- Give SQL analysts access to familiar math concepts, statistical methods, and algorithms without learning new tools and languages.
BigQuery ML

- Democratizes ML for business customers.
  - Experts in TensorFlow, scikit-learn, etc are rare.
  - Experts in SQL are far more common.

- Analyze large datasets without sampling.
  - Scale to petabytes of data

- Avoids slow, cumbersome moving of data to/from of database.
  - Learn ML models directly in BigQuery UI.
Existing Syntax:

Example 1:

```sql
CREATE PROCEDURE [dbo].[RxTrainLogitModel] (@trained_model varbinary(max) OUTPUT)
AS
BEGIN
  EXEC sp execute external script @language = N'R',
  @script = N'
## Create model
logitObj <- rxLogit(tipped ~ passenger count +
  trip distance + trip_time_in_secs + direct_distance, data =
InputDataSet)
summary(logitObj)

## Serialize model
trained_model <- as.raw(serialized(logitObj, NULL));
'...
```

Example 2:

```sql
SELECT glm('warpbreaks_dummy',
  'glm_model',
  'breaks',
  ARRAY[1.0, "wool_B", "tension_M", "tension_H"],
  'family=poisson, link=log');

SELECT w.id,
  glm_predict(
    coef,
    ARRAY[1, "wool_B", "tension_M", "tension_H"]::float8[],
    'log') AS mu
FROM warpbreaks_dummy w, glm_model m
```
BigQuery ML Syntax

- Extension of standard SQL DDLs for creating models:

```sql
{CREATE MODEL | CREATE MODEL IF NOT EXISTS | CREATE OR REPLACE MODEL}
model name
[OPTIONS(model option_list)]
[AS query_statement]

> CREATE MODEL income_model
   OPTIONS (model_type='linear_reg')
   AS SELECT state, job, income as label FROM census_data;
```
BigQuery ML Syntax

- TVFs for prediction and other model operations:

```sql
ML.PREDICT(MODEL model_name,
            {TABLE table_name | (query_statement)})

> SELECT predicted_income FROM ML.PREDICT(MODEL 'income_model',
        SELECT state, job FROM customer_data);

ML.EVALUATE(MODEL model_name
             [, {TABLE table_name | (query_statement)}]
             [, STRUCT(<T> AS threshold))])
```
Iterative Gradient Descent (IGD)

- Find \( w \) such that:
  \[
  Xw = y
  \]
  \( X \): Training data (rows: training examples, cols: features)
  \( w \): Weights
  \( y \): Observations (income in our running example)

- Core learning algorithm is gradient descent.
  - Minimizes the objective:
    \[
    \min_w \sum_i L(w^T x_i, y_i) + \lambda_1 \|w\|_1 + \lambda_2 (\|w\|_2)^2
    \]
  - Includes support for L1 and L2 regularization.
BigQuery ML Syntax

- TVFs for prediction and other model operations:
  
  ```sql
  ML.PREDICT(MODEL model_name,
              {TABLE table_name | (query_statement)})
  
  > SELECT predicted_income FROM ML.PREDICT(MODEL 'income_model',
              SELECT state, job FROM customer_data);
  
  ML.EVALUATE(MODEL model_name
              [, {TABLE table_name | (query_statement)}]
              [, STRUCT(<T> AS threshold))]
  ```
Iterative Gradient Descent (IGD)

- Find \( w \) such that:
  \[
  X w = y
  \]
  \( X \): Training data (rows: training examples, cols: features)
  \( w \): Weights
  \( y \): Observations (income in our running example)

- Core learning algorithm is gradient descent.
  - Minimizes the objective:
    \[
    \min_w \sum_i L(w^T x_i, y_i) + \lambda_1||w||_1 + \lambda_2(||w||_2)^2
    \]
    - Includes support for L1 and L2 regularization.
Iterative Gradient Descent (IGD)

- Gradient descent implemented as sequence of pure SQL queries.
- Represents data and models as tables:

  **data**

<table>
<thead>
<tr>
<th>state</th>
<th>job</th>
<th>Income</th>
</tr>
</thead>
<tbody>
<tr>
<td>NY</td>
<td>nurse</td>
<td>65000</td>
</tr>
<tr>
<td>CA</td>
<td>chef</td>
<td>55000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

  **model**

<table>
<thead>
<tr>
<th>feature</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>state:CA</td>
<td>+5.7</td>
</tr>
<tr>
<td>job:nurse</td>
<td>-3.5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- [Umar Syed, Sergei Vassilvitskii: SQML: large-scale in-database machine learning with pure SQL. SoCC 2017: 659]
Iterative Gradient Descent (IGD)

- Each algorithm iteration issues SQL queries that join model to data, update model, then write model back to disk.

```
model table

# score examples
# with model
SELECT ... FROM model JOIN data
GROUP BY example;
```

```
score table

data table

# compute gradient
# and update model
SELECT ... FROM score JOIN data
GROUP BY feature;
```

```
data table

new model table
```

- Techniques from stats and ML fields to make the queries scale and hide the complexity while achieving high accuracy.
Closed form solution

- Find $w$ such that:
  \[ X w = y \]
  $X$: Training data $\rightarrow$ $m \times n$ matrix ($m \gg n$)
  $w$: Weights $\rightarrow$ $n \times 1$
  $y$: Observations $\rightarrow$ $m \times 1$
Closed Form Solution

- ML training Algorithms mainly focus on computational linear algebra and optimizations.
- Closed form solutions expressed in matrix and vector operations.
  - Least square normal equation (linear regression)

\[ w = (X^T X + \lambda I)^{-1} X^T y \]
Closed Form Solution

- Why closed form solution is not preferred in most ML platforms?
  - Load all of the training data X into memory for computing.
  - Parallel computing of extra-large linear algebra.
    - Matrix multiplication
    - Matrix Inversion
Closed Form Solution

- ML training Algorithms mainly focus on computational linear algebra and optimizations.

- Closed form solutions expressed in matrix and vector operations.
  - Least square normal equation (linear regression)

\[ w = (X^T X + \lambda I)^{-1} X^T y \]
Closed Form Solution

- Matrix are represented as table: with schema `<row:string, col:string, data:double>`

- Matrix Multiplication is done via inner join.

```sql
SELECT
  A.row AS row,
  B.col AS col,
  SUM(A.value * B.value) AS value
FROM
  A JOIN B
ON A.col = B.row
GROUP BY
  row, col;
```
Closed Form Solution

- Matrix Inversion
  - Matrix size is $N \times N$, with $N$ number of features
    - Single shard compute
  - Symmetric Positive-Definite (SPD) Matrix
  - Fast solver (using Cholesky decomposition)
When to use IGD vs closed form?

1. If total cardinalities of training features are more than 10000, IGD strategy is used.

2. If there is overfitting issue, i.e., num of training examples is less than 10x of total cardinality, IGD is used.

3. If l1_reg or warm_start is specified, IGD strategy is used.

4. Normal equation strategy is used for all other cases.
Conclusion

- SQL analysts want to extract insight from their data
- Pure SQL works for insights from historic data
- BQML
  - Minimal ML knowledge
  - SQL syntax
  - Pure SQL implementation → Petabyte scale ML
  - In database execution
- Try it for free: https://cloud.google.com/bigquery

Acknowledgements: Thanks to the BigQuery team.
Columnar Data Storage Format: Capacitor

- Partial dictionary encoding
- RLE
- Bloom Filters
- Statistics
- Row Reordering
- Execution Pushdown

https://cloud.google.com/blog/products/gcp/inside-capacitor-bigqueries-next-generation-columnar-storage-format
Data replication

Zone A → Zone C

Zone B

Region X

Off-region
Physical Metadata

Table 1

Chunk 1  Chunk 2  Chunk 3

Colossus
Storage Optimizer

<table>
<thead>
<tr>
<th>Chunk 1 @T1</th>
<th>Chunk 2 @T2</th>
<th>Chunk 3 @T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB</td>
<td>100 kB</td>
<td>2 MB</td>
</tr>
</tbody>
</table>

Chunk 4 @T4
Provenance:
[T1-T3]
2.5 MB

Generation 0

Generation 1
Storage management

- **Data layout**
  - Columns that are often queried together - placed close to each other
  - Rows reordered to match query patterns

- **File Encoding**
  - Replicated: Faster
  - Reed Solomon: Smaller

- **Block Sizes**
  - Larger: less overhead
  - Smaller: more parallelism

- **Storage Media**
  - SSD: faster
  - HDD: less expensive
Storage Optimizer

Table

Chunk 1
@T1
1 MB

Chunk 2
@T2
100 kB

Chunk 3
@T3
2 MB

Chunk 4
@T4
Provenance:
[T1-T3]
2.5 MB

Generation 0

Generation 1
Storage management

- Data layout
  - Columns that are often queried together - placed close to each other
  - Rows reordered to match query patterns
- File Encoding
  - Replicated: Faster
  - Reed Solomon: Smaller
- Block Sizes
  - Larger: less overhead
  - Smaller: more parallelism
- Storage Media
  - SSD: faster
  - HDD: less expensive
Time Travel (FOR SYSTEM_TIME AS OF)

Table 1
Time Travel (FOR SYSTEM_TIME AS OF)

Table 1

Chunk 1 @T1

Colossus
Time Travel (FOR SYSTEM_TIME AS OF)

Table 1

Chunk 1 @T1

Chunk 2 @T2

Colossus
Time Travel (FOR SYSTEM_TIME AS OF)

Table 1

- Chunk 1 @T1
- Chunk 2 @T2

Colossus

- Chunk 3 @T3

At T3: Read as of T2.5 uses Chunk 3
At T3: Read as of T1.5 uses Chunk 1
UPDATE ... WHERE customerId = "1234"
Streaming Ingestion

- Storage
- Compute
- Petabit Network
- SQL Query
- Fast Batch Load
- Streaming Ingest
UPDATE ... WHERE customerID = "1234"
Streaming Ingestion

- Storage
- Compute
- Petabit Network
- BigQuery
- Streaming Ingest
- Fast Batch Load
- SQL Query
Streaming Ingestion

- Streaming Ingest
- Fast Batch Load
- Petabit Network
- Compute
- Storage
- BigQuery
- SQL Query