CockroachDB’s Query Optimizer

University of Washington CSEP590D, April 14, 2022
Presented by Rebecca Taft
CockroachDB: Make Data Easy

- Resilient
- Scalable
- Geo-Distributed
- SQL
- Open Source
A Real CockroachDB Deployment
Architecture of CockroachDB
Agenda

1. Intro to CockroachDB
2. Query optimization in CockroachDB
3. Generating alternative plans
4. Choosing a plan
5. Locality awareness
6. Theory vs. Practice
1. Intro to CockroachDB
2. Query optimization in CockroachDB
3. Generating alternative plans
4. Choosing a plan
5. Locality awareness
6. Theory vs. Practice
Query Optimization in CockroachDB

- Need an optimizer to support SQL
- Why not use Postgres (or other open source) optimizer?
  - CockroachDB codebase is written in go
  - Execution plans are very different in CockroachDB
  - Optimizer is key to DBMS performance
CockroachDB’s First Optimizer

- Not an optimizer
- Used heuristics (rules) to choose execution plan
- E.g. “if an index is available, always use it”
- E.g. “always use the index, except when the table is very small or we expect to scan more than 75% of the rows, or the index is located on a remote machine”
- Sort of works for OLTP, but customers run everything
CockroachDB’s Cost-Based Optimizer

- Instead of applying rigid rules, consider multiple alternatives
- Assign a cost to each alternative
- Choose lowest cost option
- Cascades-style optimization with unified search
How to generate alternatives

- Start with default plan from SQL query
- Perform a series of transformations
- Store alternatives in a compact data structure called memo
Assign cost to alternative plans

- Factors that affect cost:
  - Hardware configuration
  - Data distribution
  - Type of operators
  - Number of rows processed by each operator
Agenda

1. Intro to CockroachDB
2. Query optimization in CockroachDB
3. Generating alternative plans
4. Choosing a plan
5. Locality awareness
6. Theory vs. Practice
Phases of plan generation

Parse → Optbuild → Normalize → Explore → DistSQL Planning
Sample query

CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1;
Phases of plan generation

- Parse
- Optbuild
- Normalize
- Explore
- DistSQL Planning
SELECT * FROM ab JOIN cd ON b = c WHERE b > 1
Phases of plan generation

1. Parse
2. Optbuild
3. Normalize
4. Explore
5. DistSQL Planning
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
Optbuild: Semantic analysis

“The angry toaster oven praises the discovery.”
“The tall sky talks to the plant of my soul.”

```
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
```

- Are `ab` and `cd` real tables in the database that current user has permissions to read?
- Do columns `b` and `c` exist in tables `ab` and `cd`, and are they unique?
- What columns are selected by ‘*’?
Phases of plan generation

1. Parse
2. Optbuild
3. Normalize
4. Explore
5. DistSQL Planning
Normalization

Optbuilder thinks it’s constructing this:

```
project
└── select
    ├── inner-join (hash)
    │    ├── scan ab
    │    ├── scan cd
    │    └── filters
    │         └── b = c
    └── filters
         └── b > 1
```

But it actually constructs this:

```
inner-join (hash)
├── select
│    ├── scan ab
│    └── filters
│         └── b > 1
├── select
│    └── scan cd
│         └── filters
│                  └── c > 1
└── filters
     └── b = c
```
Normalization rules

- Transformation rules create a logically equivalent relational expression
- Normalization (or “rewrite”) rules are “always a good idea” to apply
- Examples
  - Eliminate unnecessary operations: \( \text{NOT (NOT } x \text{)} \rightarrow x \)
  - Canonicalize expressions: \( 5 = x \rightarrow x = 5 \)
  - Constant folding: \( \text{length(‘abc’)} \rightarrow 3 \)
  - Predicate push-down*
  - De-correlation of subqueries*
  - ...

* Not always a good idea, but almost always
# EliminateNot discards a doubled Not operator.

[EliminateNot, Normalize]

(Not (Not $input:*))

=>

$input
DSL: Optgen

// ConstructNot constructs an expression for the Not operator.

    // [EliminateNot]
    {
        _not, _ := input.(*memo.NotExpr)
        if _not != nil {
            input := _not.Input
            if _f.matchedRule == nil || _f.matchedRule(opt.EliminateNot) {
                _expr := input
                return _expr
            }
        }
    }

    // ... other rules ...

    e := _f.mem.MemoizeNot(input)
    return _f.onConstructScalar(e)
}
# MergeSelects combines two nested Select operators into a single Select that
# ANDs the filter conditions of the two Selects.

[MergeSelects, Normalize]

(Select (Select $input:* $innerFilters:* $filters:*))
=>
(Select $input (ConcatFilters $innerFilters $filters:*))
DSL: Optgen

```go
// [MergeSelects]
{
    _select, _ := input.(*memo.SelectExpr)
    if _select != nil {
        input := _select.Input
        innerFilters := _select.Filters
        if _f.matchedRule == nil || _f.matchedRule(opt.MergeSelects) {
            _expr := _f.ConstructSelect(
                input,
                _f.funcs.ConcatFilters(innerFilters, filters),
            )
            return _expr
        }
    }
}
```
Phases of plan generation

1. Parse
2. Optbuild
3. Normalize
4. Explore
5. DistSQL Planning
Exploration

- Exploration rules may or may not produce a better plan
- Examples:
  - Join reordering: \( A \) join \( (B \) join \( C) \) \( \rightarrow \) \( (A \) join \( B) \) join \( C \)
  - Join algorithm (e.g., hash join, merge join, lookup join…)
  - Index selection
Memo after normalization

* scalar expressions are omitted but are also groups (always single-element)

CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
**Explore: GenerateIndexScans**

CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
Explore: GenerateConstrainedScans

CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
Explore: GenerateConstrainedScans

CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
Explore: Reorder Joins

```
CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
```
**Explore: GenerateMergeJoins**

<table>
<thead>
<tr>
<th>Group</th>
<th>Operation</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>InnerJoin</td>
<td><code>b=c</code></td>
</tr>
<tr>
<td></td>
<td>MergeJoin</td>
<td><code>b,c</code></td>
</tr>
<tr>
<td>2</td>
<td>Select</td>
<td><code>b&gt;1</code></td>
</tr>
<tr>
<td></td>
<td>Scan</td>
<td><code>ab@b</code></td>
</tr>
<tr>
<td>3</td>
<td>Select</td>
<td><code>c&gt;1</code></td>
</tr>
<tr>
<td></td>
<td>Scan</td>
<td><code>cd</code></td>
</tr>
<tr>
<td>4</td>
<td>Scan</td>
<td><code>ab@primary</code></td>
</tr>
<tr>
<td>5</td>
<td>Scan</td>
<td><code>cd@primary</code></td>
</tr>
</tbody>
</table>

CREATE TABLE `ab` (
  `a` INT PRIMARY KEY,
  `b` INT,
  INDEX (`b`));

CREATE TABLE `cd` (
  `c` INT PRIMARY KEY,
  `d` INT);

SELECT * FROM `ab` JOIN `cd` ON `b`=`c` WHERE `b`>1
Explore: GenerateLookupJoins

CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
Explore: GenerateLookupJoins

```
CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
```
Explore: GenerateLookupJoins

CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
Phases of plan generation

Parse → Optbuild → Normalize → Explore → DistSQL Planning
DistSQL Planning
1. Intro to CockroachDB
2. Query optimization in CockroachDB
3. Generating alternative plans
4. Choosing a plan
5. Locality awareness
6. Theory vs. Practice
Assign cost to alternative plans

- Factors that affect cost:
  - Hardware configuration
  - Data distribution
  - Type of operators
  - Number of rows processed by each operator
Assign cost to alternative plans

- Factors that affect cost:
  - Hardware configuration
  - Data distribution
  - Type of operators
  - Number of rows processed by each operator
Find number of rows processed

● Use statistics
● Collect statistics on each table:
  ○ Row count
  ○ Distinct count
  ○ Null count
  ○ Histogram
● Estimate how stats change as data flows through execution plan
CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
Multi-column stats

- Improves stats for correlated columns
  - E.g., WHERE state = ‘Illinois’ AND city = ‘Chicago’
- Which sets of columns to use? $2^n$ possibilities...
- Use index prefixes
  - For index on (a, b, c), collect multi-column stats on (a, b) and (a, b, c)
  - Currently only distinct and null counts
  - Multi-column histograms coming later
CREATE STATISTICS
Agenda

1. Intro to CockroachDB
2. Query optimization in CockroachDB
3. Generating alternative plans
4. Choosing a plan
5. Locality awareness
6. Theory vs. Practice
Locality-Aware SQL Optimization and Execution

Network latencies and throughput are important considerations in geo-distributed setups.

Historically required expert users to shard and place data in specific regions.
Locality-Aware SQL Optimization and Execution

Database should be aware of regions, so users don’t need to be.

New concept: Table Locality
REGIONAL or GLOBAL

Tables accessed from a single region or amenable to partitioning use locality REGIONAL

Read-mostly tables not amenable to partitioning use locality GLOBAL

Queries leverage data closest to them
Example: movr application

### users

<table>
<thead>
<tr>
<th>email</th>
<th>home_addr...</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:joe@excite.com">joe@excite.com</a></td>
<td>123 Ash St</td>
</tr>
<tr>
<td><a href="mailto:jane@gmail.com">jane@gmail.com</a></td>
<td>55 Main St</td>
</tr>
</tbody>
</table>

### promo_codes

<table>
<thead>
<tr>
<th>code</th>
<th>discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVE$$</td>
<td>15</td>
</tr>
<tr>
<td>FREE</td>
<td>100</td>
</tr>
</tbody>
</table>

SELECT * FROM users WHERE email = 'joe@excite.com';

SELECT * FROM promo_codes WHERE code = 'SAVE$$';
**Example: movr application**

### users

<table>
<thead>
<tr>
<th>region</th>
<th>email</th>
<th>home_addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>us-west</td>
<td><a href="mailto:joe@excite.com">joe@excite.com</a></td>
<td>123 A...</td>
</tr>
<tr>
<td>us-east</td>
<td><a href="mailto:jane@gmail.com">jane@gmail.com</a></td>
<td>55 Ma...</td>
</tr>
<tr>
<td>us-east</td>
<td><a href="mailto:joe@excite.com">joe@excite.com</a></td>
<td>99 Lo...</td>
</tr>
</tbody>
</table>

### promo_codes

<table>
<thead>
<tr>
<th>code</th>
<th>discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVE$$</td>
<td>15</td>
</tr>
<tr>
<td>FREE</td>
<td>100</td>
</tr>
</tbody>
</table>
Example: movr application

SELECT * FROM users WHERE email = 'joe@excite.com';

SELECT * FROM users WHERE email = 'joe@gmail.com';

SELECT * FROM users WHERE email = 'jane@gmail.com';

SELECT * FROM promo_codes WHERE code = 'SAVE$$';

<table>
<thead>
<tr>
<th>email</th>
<th>home_addr</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:joe@excite.com">joe@excite.com</a></td>
<td>123 Ash St</td>
<td></td>
</tr>
<tr>
<td><a href="mailto:joe@gmail.com">joe@gmail.com</a></td>
<td>99 Long Ln</td>
<td></td>
</tr>
<tr>
<td><a href="mailto:jane@gmail.com">jane@gmail.com</a></td>
<td>55 Main St</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>code</th>
<th>discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVE$$</td>
<td>15</td>
</tr>
<tr>
<td>FREE</td>
<td>100</td>
</tr>
</tbody>
</table>
Regional tables

In REGIONAL BY ROW, data is partitioned by a hidden crdb_region column, which is set to the local region on insert.

Post-query uniqueness checks ensure that email remains unique.
> EXPLAIN (OPT) INSERT INTO users (email, name)
VALUES ('becca@cockroachlabs.com', 'Rebecca Taft');

info

insert users
└── values
    └── ('becca@cockroachlabs.com', 'Rebecca Taft',
        gen_random_uuid(), 'us-west1')

unique-check: users(email)
└── semi-join (lookup users@users_email_key)
    └── filters
        (id != users.id) OR
        (crdb_region != users.crdb_region)
Reading from a Regional by Row table

Automatically checks the local region first before fanning out to remote regions

CREATE TABLE users (  
id UUID PRIMARY KEY DEFAULT gen_random_uuid()  
email STRING UNIQUE,  
name STRING  
) LOCALITY REGIONAL BY ROW

SELECT * FROM users WHERE  
email = 'joe@excite.com';
Reading from a Regional by Row table

```sql
> EXPLAIN (OPT) SELECT * FROM users
WHERE email = 'becca@cockroachlabs.com';
```

```
info

------------------------------------------------------------
- index-join users
  └── locality-optimized-search
    └── scan users@users_email_key
      └── [/'us-west1'/'becca@cockroachlabs.com']
    └── scan users@users_email_key
      └── [/'europe-west1'/'becca@cockroachlabs.com']
      └── [/'us-east1'/'becca@cockroachlabs.com']
```
Global tables

Non-voting replicas which don’t impact write latency

System automatically places a non-voting replica in regions without a voting replica

“Global transactions” cause writes to commit at a future timestamp and avoid blocking reads

CREATE TABLE promo_codes (  
  code STRING,  
  discount FLOAT  
) LOCALITY GLOBAL
Local reads from Global tables

Automatically reads from replica (voting or non-voting) in the read’s region

CREATE TABLE promo_codes (  
  code STRING,  
  discount FLOAT  
) LOCALITY GLOBAL

SELECT * FROM promo_codes  
WHERE code = 'SAVE$$';
Locality-Aware SQL Optimization: What’s Next?

- Move DistSQL planning into optimizer
- Incorporate latency into cost model
- Add transformations using foreign key relationships to constrain plan to a single region
  - Add a join between a GLOBAL and REGIONAL BY ROW table with an FK relationship
Agenda

1. Intro to CockroachDB
2. Query optimization in CockroachDB
3. Generating alternative plans
4. Choosing a plan
5. Locality awareness
6. Theory vs. Practice
Optimizing for OLTP

- Focus on minimizing overhead for simple OLTP queries (e.g., primary key lookup)
- Logical properties essential for optimization
  - Cardinality (different from stats)
  - Functional dependencies
  - Not-null columns
  - ...
- As of today, 268 normalization rules, 41 exploration rules
- Foreign key checks and cascades optimized as “post queries”
Join Ordering

- Almost shipped v1 without join ordering
- Initially implemented with two rules: CommuteJoin and AssociateJoin
- Reordered at most 4 tables by default
- CockroachDB now uses implementation of DPSUBE
- Now orders up to 8 tables by default
Query Cache

- LRU cache keyed on SQL string
- Stores optimized memo
- For prepared statements w/ placeholders, stores normalized memo
- To execute prepared statement:
  - Replace placeholders in normalized memo
  - Perform additional normalization and exploration
Optimizer hints

To force specific index, add @index_name to table name:

```
EXPLAIN SELECT * FROM ab @b WHERE a=1;
```

---

To force specific join type, add HASH / MERGE / LOOKUP between INNER / LEFT / RIGHT / FULL and JOIN:

```
EXPLAIN SELECT * FROM ab INNER HASH JOIN cd ON a=c;
```
EXPLAIN ANALYZE (DEBUG) SELECT ...

Statement diagnostics bundle generated. Download from the Admin UI (Advanced Debug -> Statement Diagnostics History), via the direct link below, or using the command line.
Admin UI: http://127.0.0.1:57782
Direct link: http://127.0.0.1:57782/_admin/v1 stmtbundle/585176264079081475
Command line: cockroach statement-diag list / download
(6 rows)

Time: 65.133ms
### Debugging tools

**EXPLAIN ANALYZE (DEBUG) SELECT ...**

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Kind</th>
<th>Date Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>stmt-bundle-585176264079081475</td>
<td>--</td>
<td>Folder</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
<tr>
<td>stats.sql</td>
<td>646 KB</td>
<td>Visual...ocument</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
<tr>
<td>schema.sql</td>
<td>1 KB</td>
<td>Visual...ocument</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
<tr>
<td>env.sql</td>
<td>249 bytes</td>
<td>Visual...ocument</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
<tr>
<td>trace-jaeger.json</td>
<td>22 KB</td>
<td>JSON Document</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
<tr>
<td>trace.txt</td>
<td>8 KB</td>
<td>Plain Text</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
<tr>
<td>trace.json</td>
<td>15 KB</td>
<td>JSON Document</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
<tr>
<td>distsql.html</td>
<td>671 bytes</td>
<td>HTML</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
<tr>
<td>plan.txt</td>
<td>830 bytes</td>
<td>Plain Text</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
<tr>
<td>opt-vv.txt</td>
<td>1 KB</td>
<td>Plain Text</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
<tr>
<td>opt-v.txt</td>
<td>896 bytes</td>
<td>Plain Text</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
<tr>
<td>opt.txt</td>
<td>175 bytes</td>
<td>Plain Text</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
<tr>
<td>statement.txt</td>
<td>68 bytes</td>
<td>Plain Text</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
<tr>
<td>stmt-bundle-585176264079081475.zip</td>
<td>61 KB</td>
<td>ZIP archive</td>
<td>Aug 28, 2020 at 5:22</td>
</tr>
</tbody>
</table>
Summary

1. Intro to CockroachDB
2. Query optimization in CockroachDB
3. Generating alternative plans
4. Choosing a plan
5. Locality awareness
6. Theory vs. Practice
Thank you

We are hiring! www.cockroachlabs.com/careers
github.com/cockroachdb/cockroach
becca@cockroachlabs.com