Cockroach Labs

CockroachDB's Query Optimizer

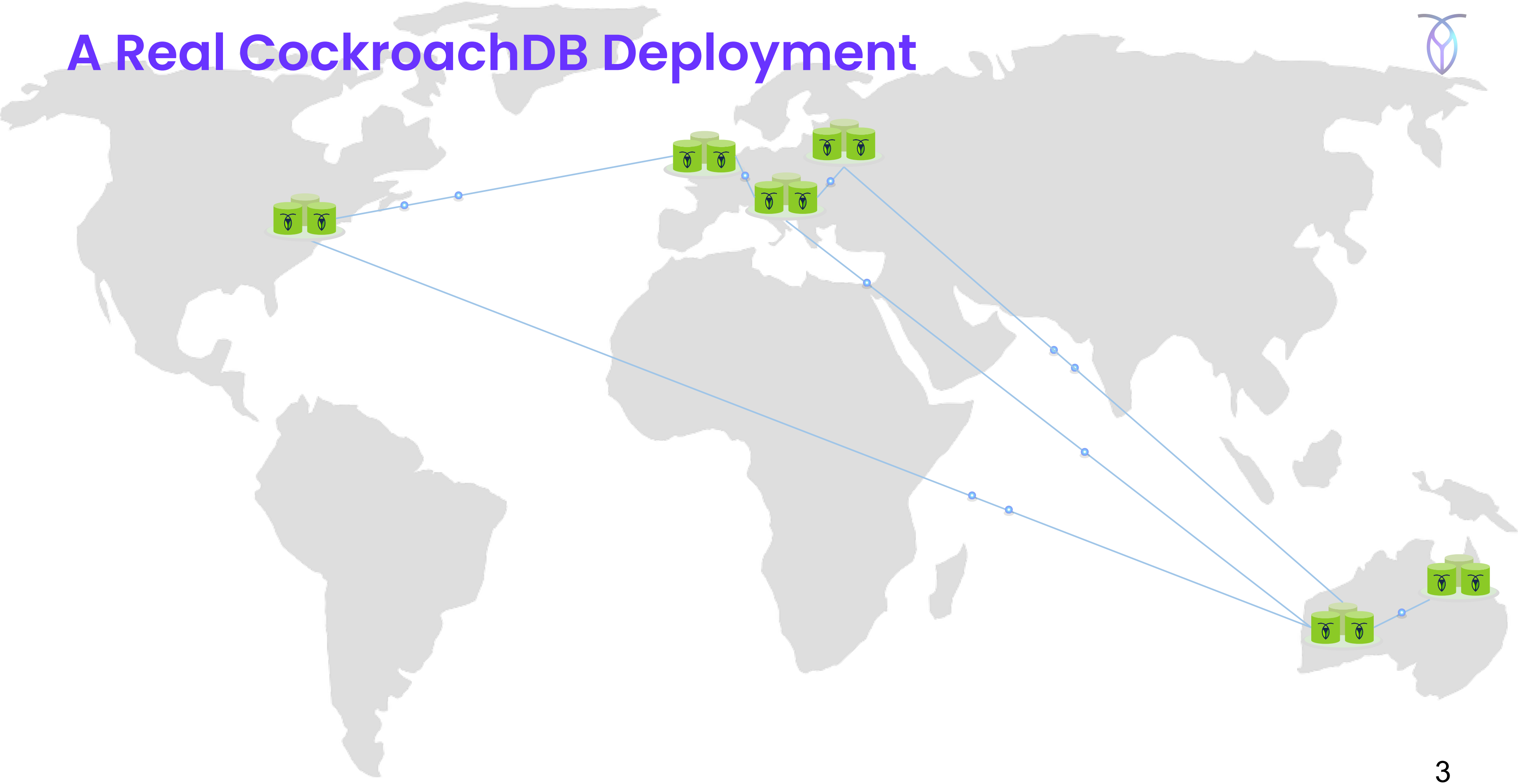
University of Washington CSEP590D, April 14, 2022

Presented by Rebecca Taft

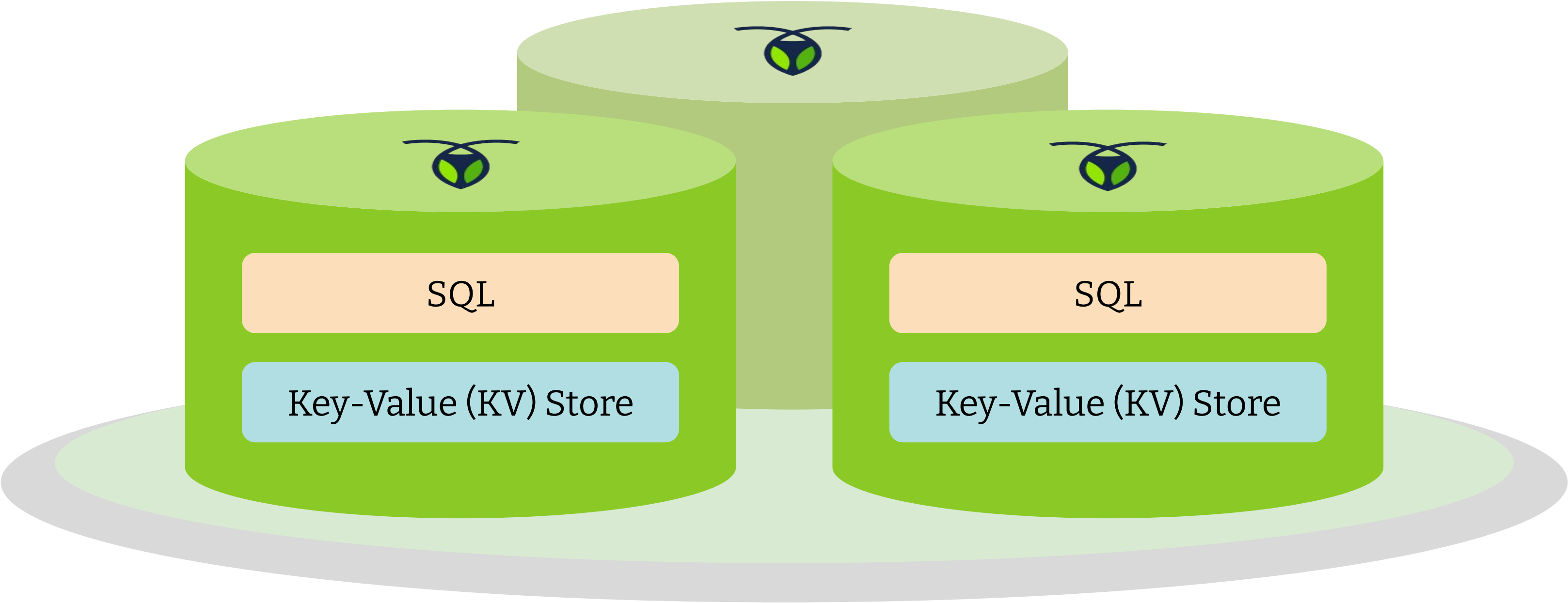
CockroachDB: Make Data Easy

- Resilient
- Scalable
- Geo-Distributed
- SQL
- Open Source

A Real CockroachDB Deployment



Architecture of CockroachDB



SQL

Key-Value (KV) Store

SQL

Key-Value (KV) Store

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

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

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

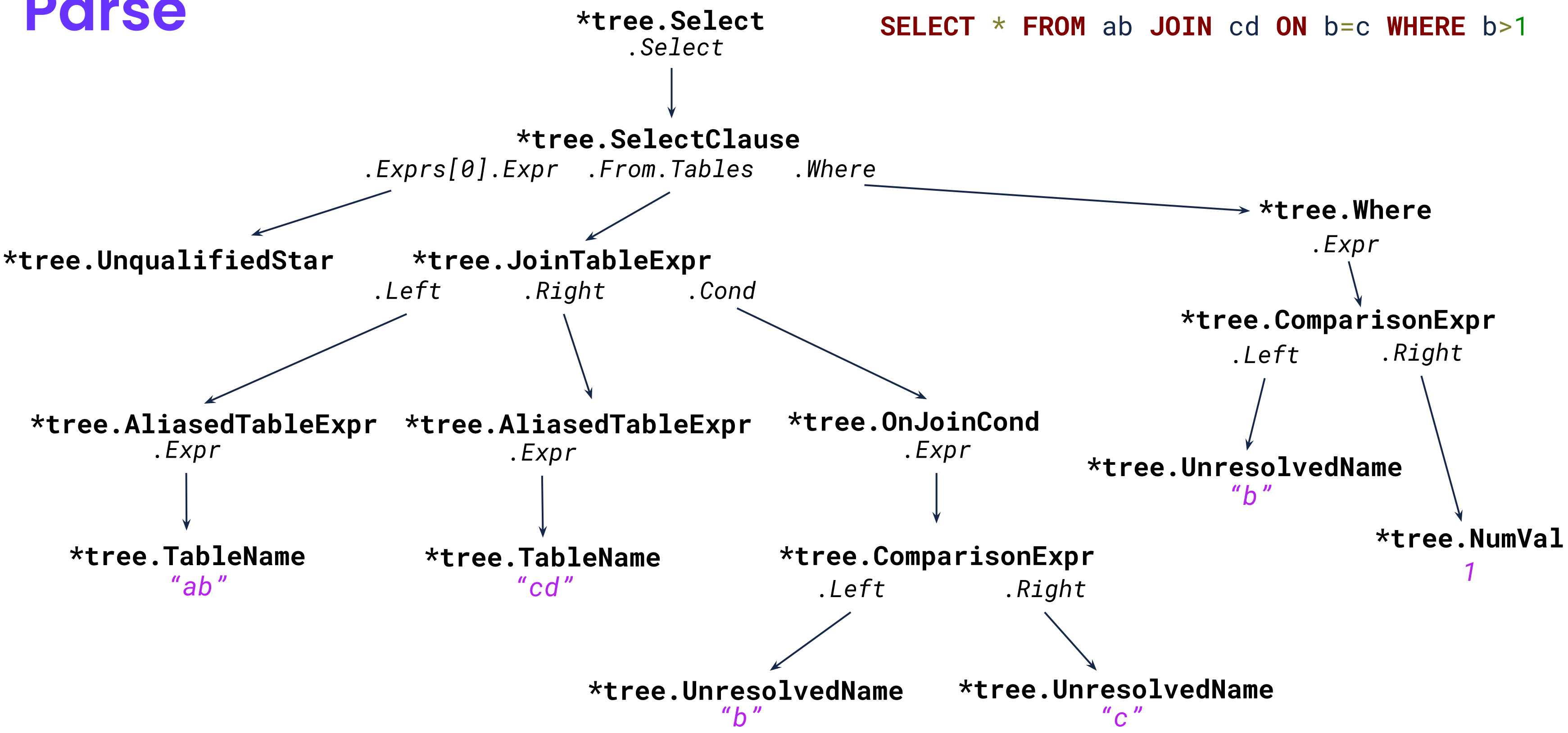
Normalize

Explore

DistSQL Planning

Parse

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



`*tree.Select`
.Select

`*tree.SelectClause`
.Exprs[0].Expr *.From.Tables* *.Where*

`*tree.UnqualifiedStar`

`*tree.JoinTableExpr`
.Left *.Right* *.Cond*

`*tree.Where`
.Expr

`*tree.ComparisonExpr`
.Left *.Right*

`*tree.AliasedTableExpr`
.Expr

`*tree.AliasedTableExpr`
.Expr

`*tree.OnJoinCond`
.Expr

`*tree.UnresolvedName`
"b"

`*tree.NumVal`
1

`*tree.TableName`
"ab"

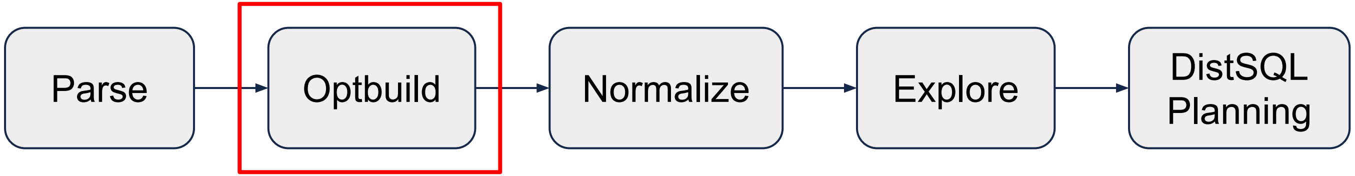
`*tree.TableName`
"cd"

`*tree.ComparisonExpr`
.Left *.Right*

`*tree.UnresolvedName`
"b"

`*tree.UnresolvedName`
"c"

Phases of plan generation



Parse

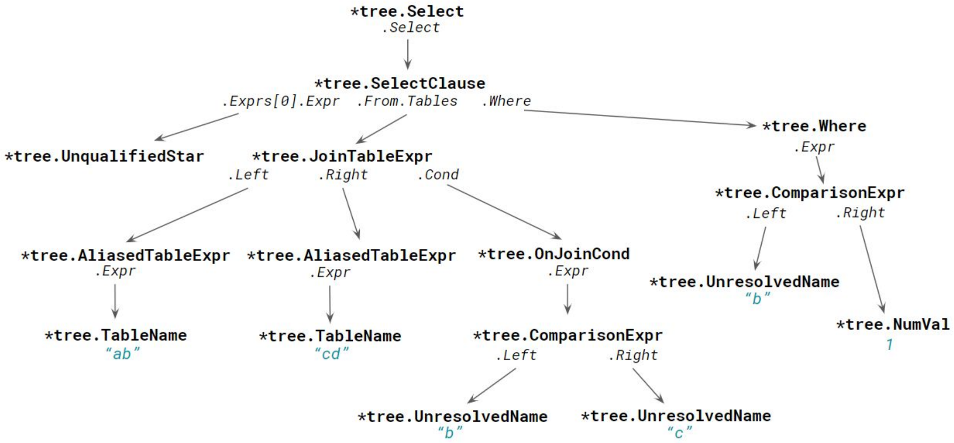
Optbuild

Normalize

Explore

DistSQL Planning

Optbuild



*tree.Select
.Select

*tree.SelectClause
.Exprs[0].Expr .From.Tables .Where

*tree.UnqualifiedStar

*tree.JoinTableExpr
.Left .Right .Cond

*tree.Where
.Expr

*tree.ComparisonExpr
.Left .Right

*tree.AliasedTableExpr
.Expr

*tree.AliasedTableExpr
.Expr

*tree.OnJoinCond
.Expr

*tree.UnresolvedName
"b"

*tree.NumVal
1

*tree.TableName
"ab"

*tree.TableName
"cd"

*tree.ComparisonExpr
.Left .Right

*tree.UnresolvedName
"b"

*tree.UnresolvedName
"c"

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

```
ConstructSelect(
  ConstructInnerJoin(
    ConstructScan(),
    ConstructScan(),
    ConstructFiltersItem(
      ConstructEq(
        ConstructVariable(),
        ConstructVariable(),
      ),
    ),
  ),
  ConstructFiltersItem(
    ConstructGt(
      ConstructVariable(),
      ConstructConst(),
    ),
  ),
)
```

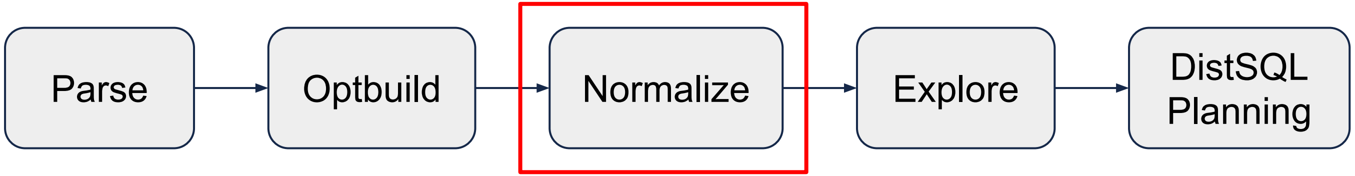
Optbuild: Semantic analysis

“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


Parse

Optbuild

Normalize

Explore

DistSQL Planning

Normalization

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

```
ConstructSelect(
  ConstructInnerJoin(
    ConstructScan(),
    ConstructScan(),
    ConstructFiltersItem(
      ConstructEq(
        ConstructVariable(),
        ConstructVariable(),
      ),
    ),
  ),
  ConstructFiltersItem(
    ConstructGt(
      ConstructVariable(),
      ConstructConst(),
    ),
  ),
)
```

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: `NOT (NOT x) -> x`
  - Canonicalize expressions: `5 = x -> x = 5`
  - Constant folding: `length(‘abc’) -> 3`
  - Predicate push-down*
  - De-correlation of subqueries*
  - ...

* Not always a good idea, but almost always

DSL: Optgen

```
# EliminateNot discards a doubled Not operator.
[EliminateNot, Normalize]
(Not (Not $input:*))
=>
$input
```

DSL: Optgen

```
// ConstructNot constructs an expression for the Not operator.
func (_f *Factory) ConstructNot(input opt.ScalarExpr) opt.ScalarExpr {

  // [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)
}
```

DSL: Optgen

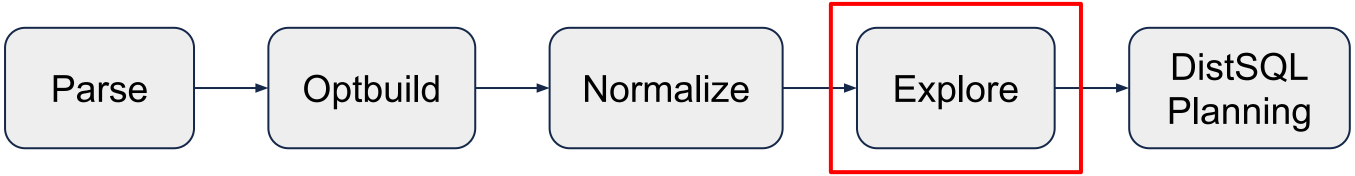
```
# MergeSelects combines two nested Select operators into a single Select that
# ANDs the filter conditions of the two Selects.
[MergeSelects, Normalize]
(Select (Select $input:* $innerFilters:*) $filters:*)
=>
(Select $input (ConcatFilters $innerFilters $filters))
```

DSL: Optgen

```
// [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
    }
  }
}
```

Invoke
custom function

Phases of plan generation



Parse

Optbuild

Normalize

Explore

DistSQL Planning

Exploration

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

Memo after normalization

Group ❶	InnerJoin ❷ ❸ b=c
Group ❷	Select ❹ b>1
Group ❸	Select ❺ c>1
Group ❹	Scan ab@primary
Group ❺	Scan cd@primary

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

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

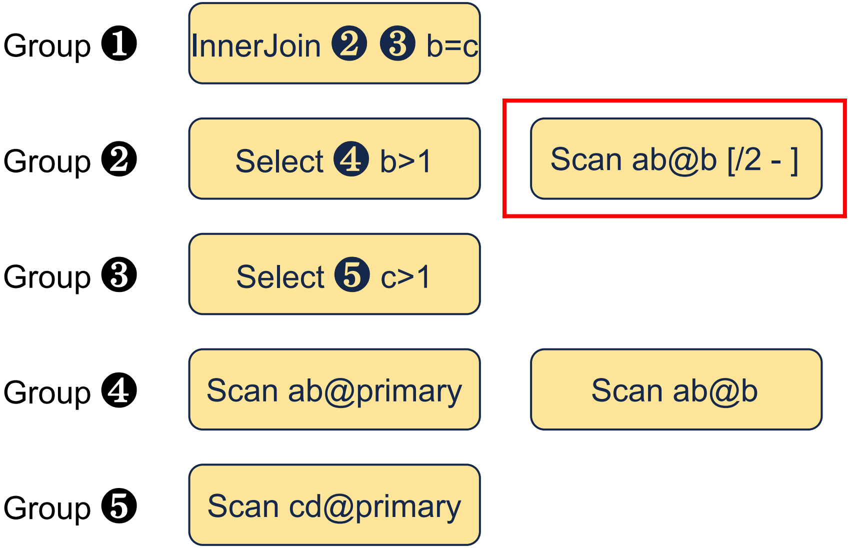
```
CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
```

Explore: GenerateIndexScans

Group ❶	InnerJoin ❷ ❸ b=c	
Group ❷	Select ❹ b>1	
Group ❸	Select ❺ c>1	
Group ❹	Scan ab@primary	Scan ab@b
Group ❺	Scan cd@primary	

```
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
```

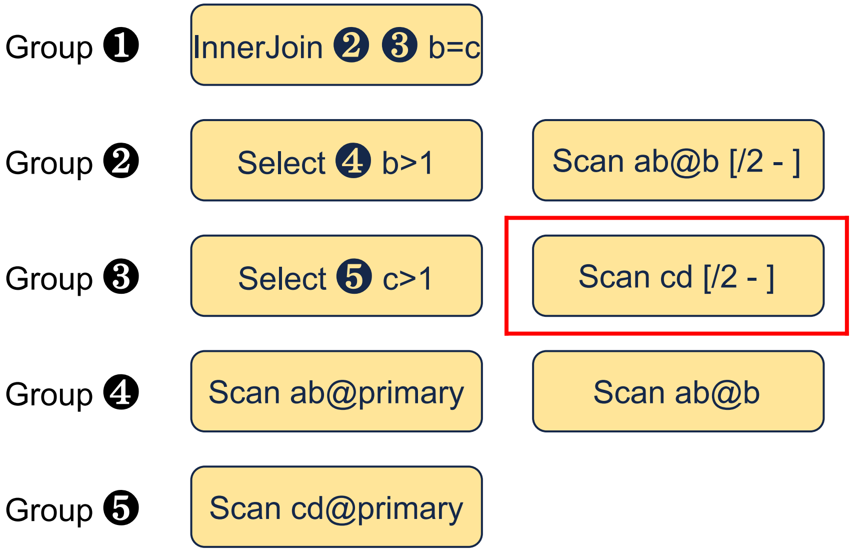
Explore: GenerateConstrainedScans



Group ❶	InnerJoin ❷ ❸ b=c	
Group ❷	Select ❹ b>1	Scan ab@b [/2 -]
Group ❸	Select ❺ c>1	
Group ❹	Scan ab@primary	Scan ab@b
Group ❺	Scan cd@primary	

```
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



Group ❶ InnerJoin ❷ ❸ b=c

Group ❷ Select ❹ b>1 Scan ab@b [/2 -]

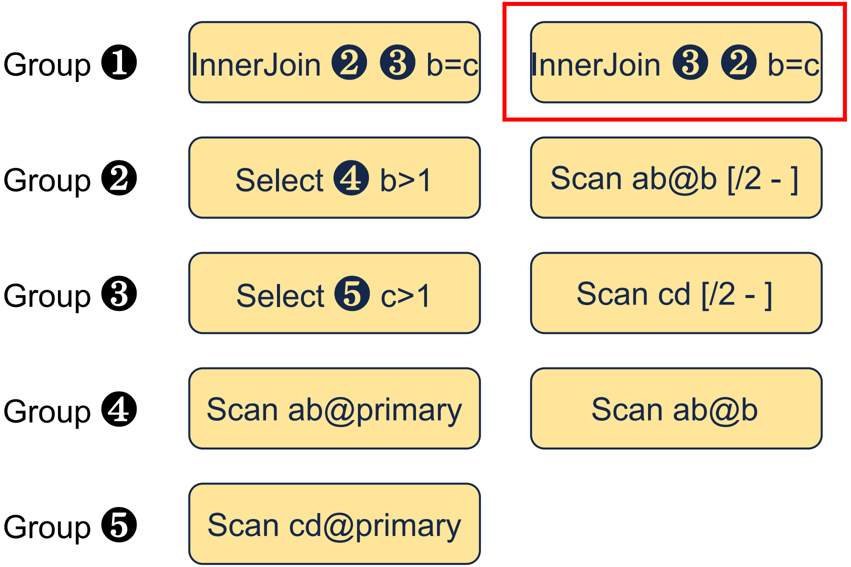
Group ❸ Select ❺ c>1 Scan cd [/2 -]

Group ❹ Scan ab@primary Scan ab@b

Group ❺ Scan cd@primary

```sql
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: ReorderJoins



Group ❶	InnerJoin ❷ ❸ b=c	InnerJoin ❸ ❷ b=c
Group ❷	Select ❹ b>1	Scan ab@b [/2 -]
Group ❸	Select ❺ c>1	Scan cd [/2 -]
Group ❹	Scan ab@primary	Scan ab@b
Group ❺	Scan cd@primary	

```
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

Group ❶	InnerJoin ❷ ❸ b=c	InnerJoin ❸ ❷ b=c	MergeJoin ❷ ❸ b,c
Group ❷	Select ❹ b>1	Scan ab@b [/2 -]	
Group ❸	Select ❺ c>1	Scan cd [/2 -]	
Group ❹	Scan ab@primary	Scan ab@b	
Group ❺	Scan cd@primary		

```sql
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

Group ❶	InnerJoin ❷ ❸ b=c	InnerJoin ❸ ❷ b=c	MergeJoin ❷ ❸ b,c	LookupJoin ❷ cd	…
Group ❷	Select ❹ b>1	Scan ab@b [/2 -]			
Group ❸	Select ❺ c>1	Scan cd [/2 -]			
Group ❹	Scan ab@primary	Scan ab@b			
Group ❺	Scan cd@primary				

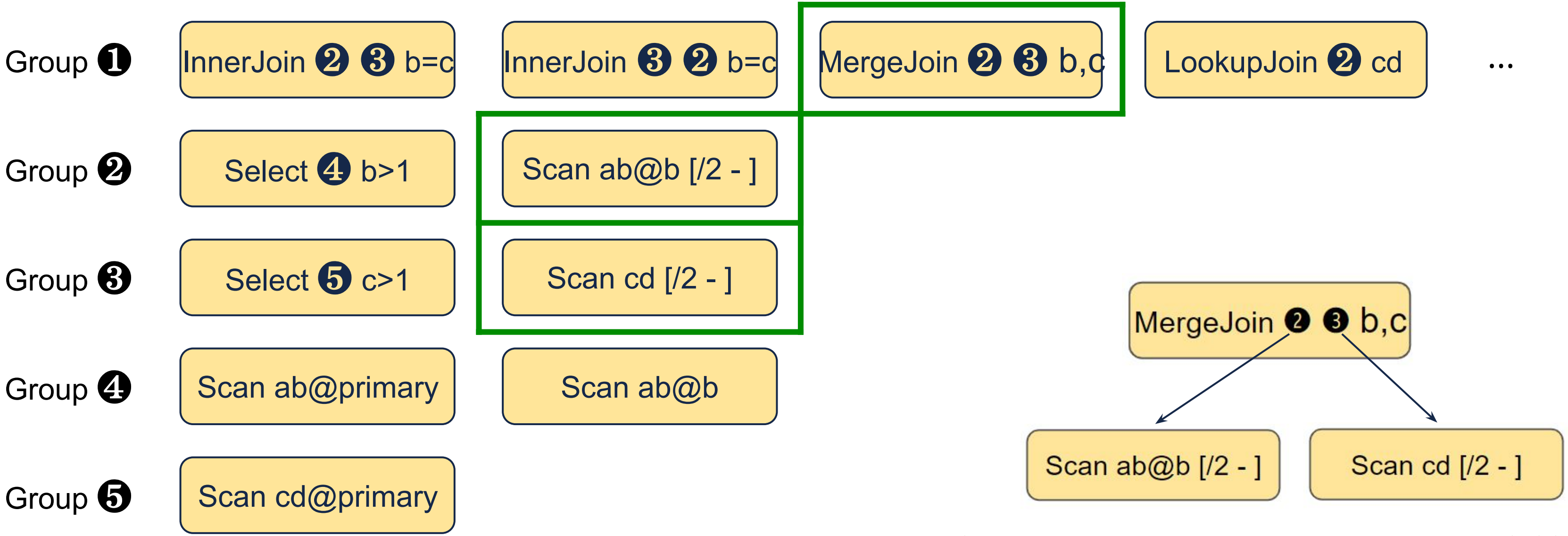
```
CREATE TABLE ab (a INT PRIMARY KEY, b INT, INDEX (b));
CREATE TABLE cd (c INT PRIMARY KEY, d INT);
SELECT * FROM ab JOIN cd ON b=c WHERE b>1
```

Explore: GenerateLookupJoins

Group ❶	InnerJoin ❷ ❸ b=c	InnerJoin ❸ ❷ b=c	MergeJoin ❷ ❸ b,c	LookupJoin ❷ cd	…
Group ❷	Select ❹ b>1	Scan ab@b [/2 -]			
Group ❸	Select ❺ c>1	Scan cd [/2 -]			
Group ❹	Scan ab@primary	Scan ab@b			
Group ❺	Scan cd@primary				

```
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
```

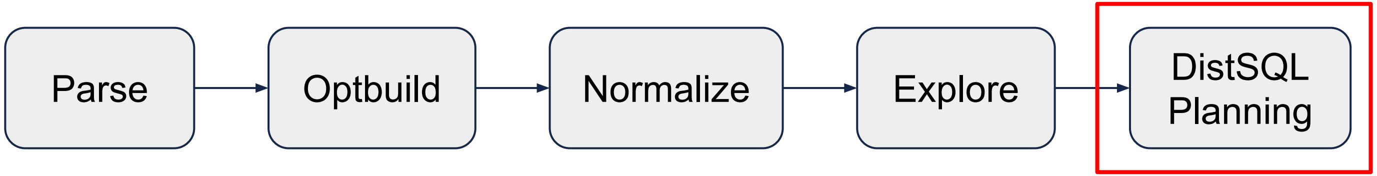
Explore: GenerateLookupJoins



Group ❶	InnerJoin ❷ ❸ b=c	InnerJoin ❸ ❷ b=c	MergeJoin ❷ ❸ b,c	LookupJoin ❷ cd	…
Group ❷	Select ❹ b>1	Scan ab@b [/2 -]			
Group ❸	Select ❺ c>1	Scan cd [/2 -]			
Group ❹	Scan ab@primary	Scan ab@b			
Group ❺	Scan cd@primary				

```
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

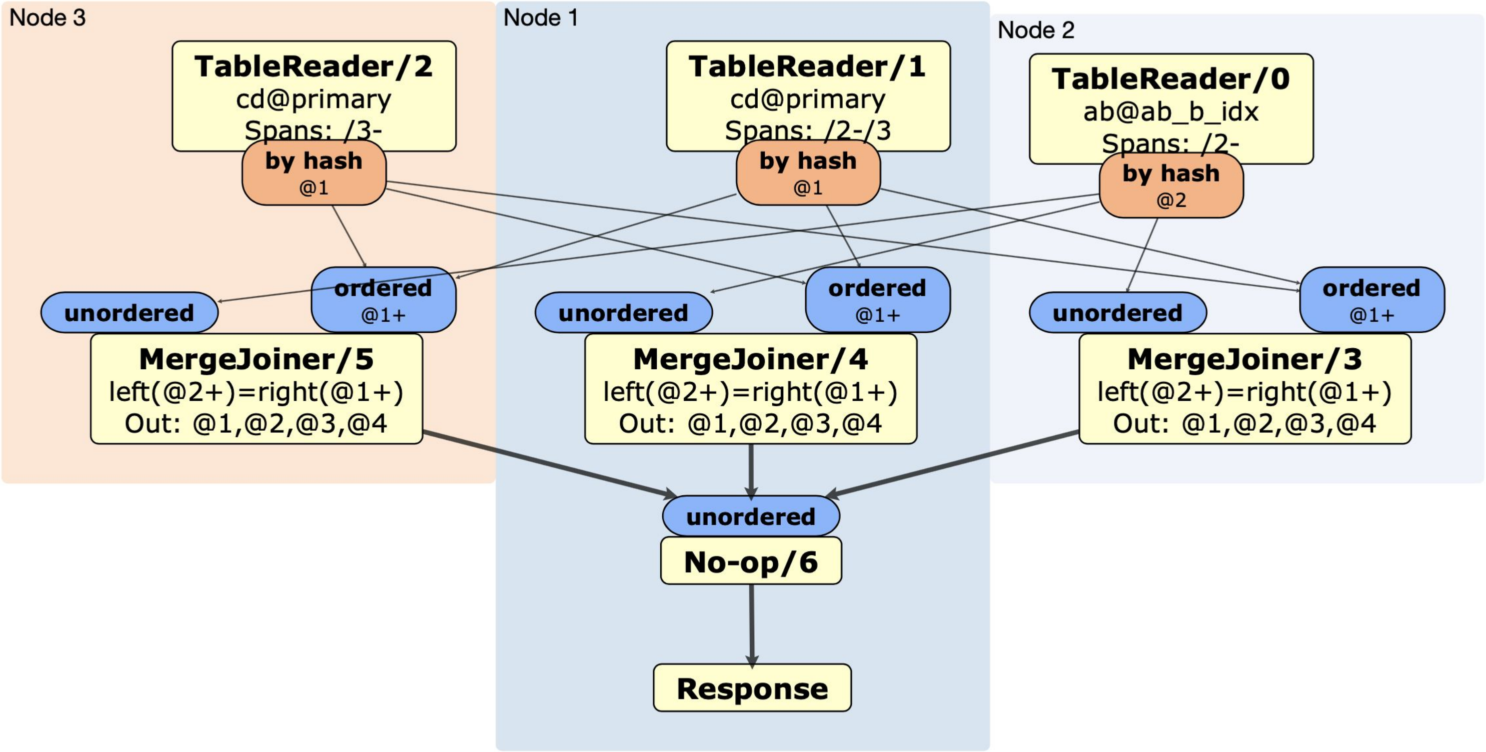
Optbuild

Normalize

Explore

DistSQL Planning

DistSQL Planning



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

Assign cost to alternative plans

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

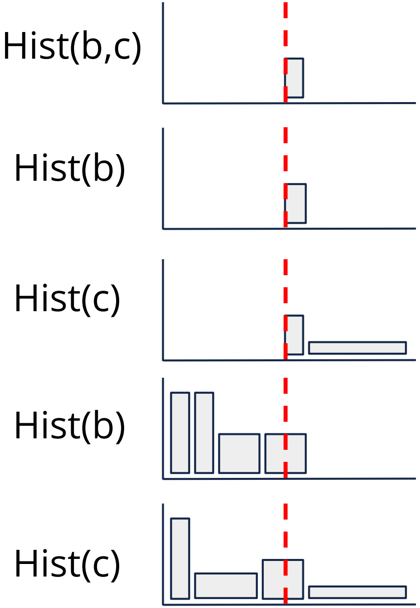
Assign cost to alternative plans

- 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

Calculate Statistics



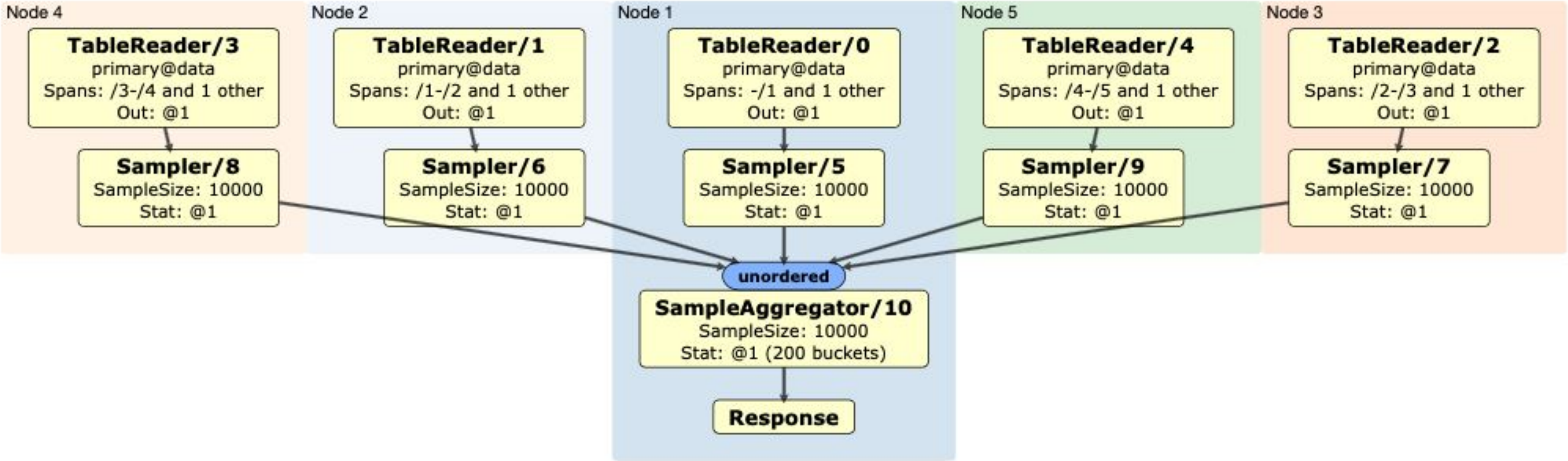
Rows: 500	Group ❶	InnerJoin ❷ ❸ b=c	InnerJoin ❸ ❷ b=c	MergeJoin ❷ ❸ b,c	…
Rows: 500	Group ❷	Select ❹ b>1	Scan ab@b [/2 -]		
Rows: 1500	Group ❸	Select ❺ c>1	Scan cd [/2 -]		
Rows: 4000	Group ❹	Scan ab@primary	Scan ab@b		
Rows: 4000	Group ❺	Scan cd@primary			

```
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



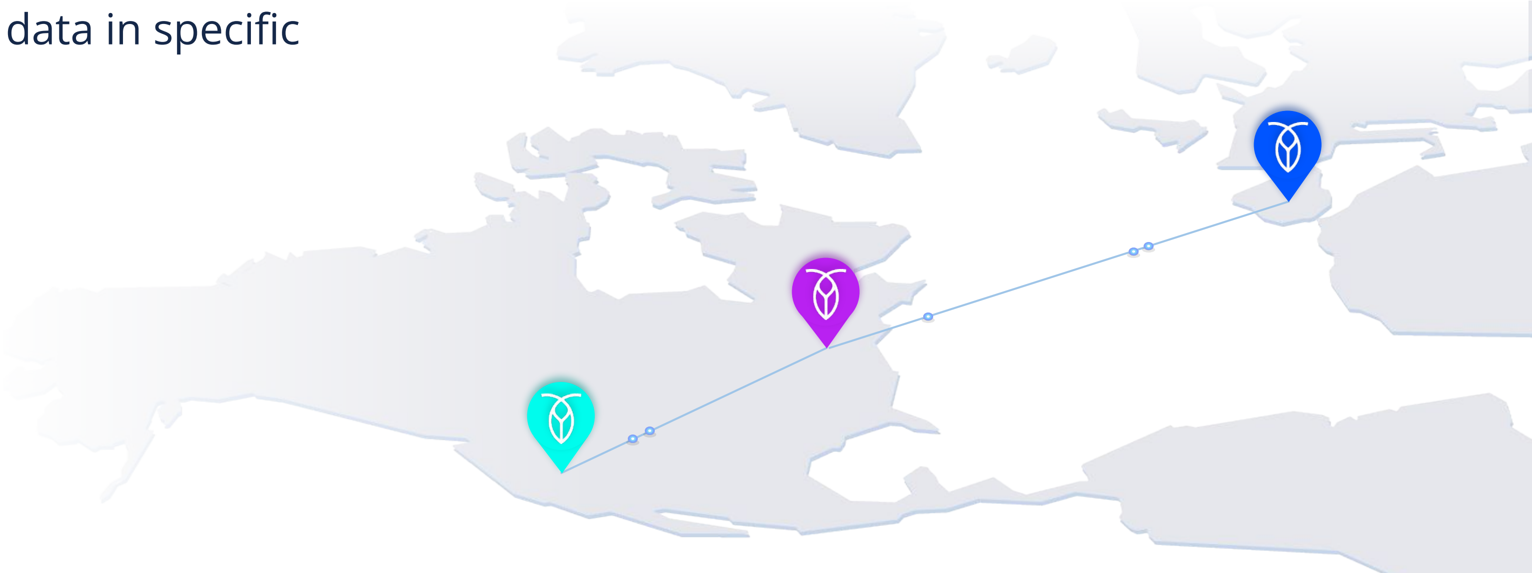
Agenda

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.

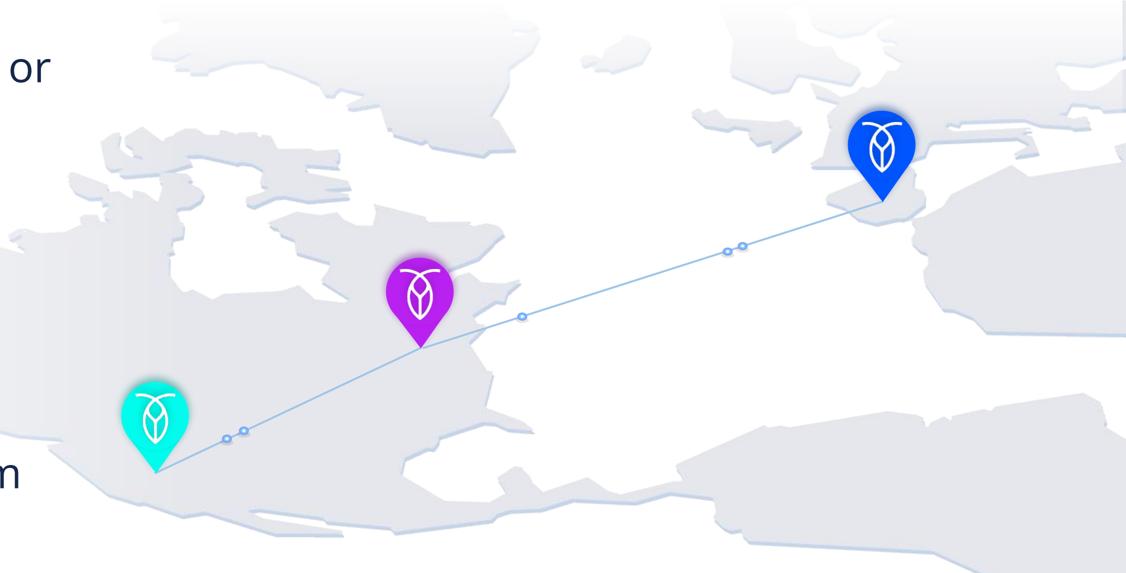
New concept: Table Locality

`REGIONAL` or `GLOBAL`

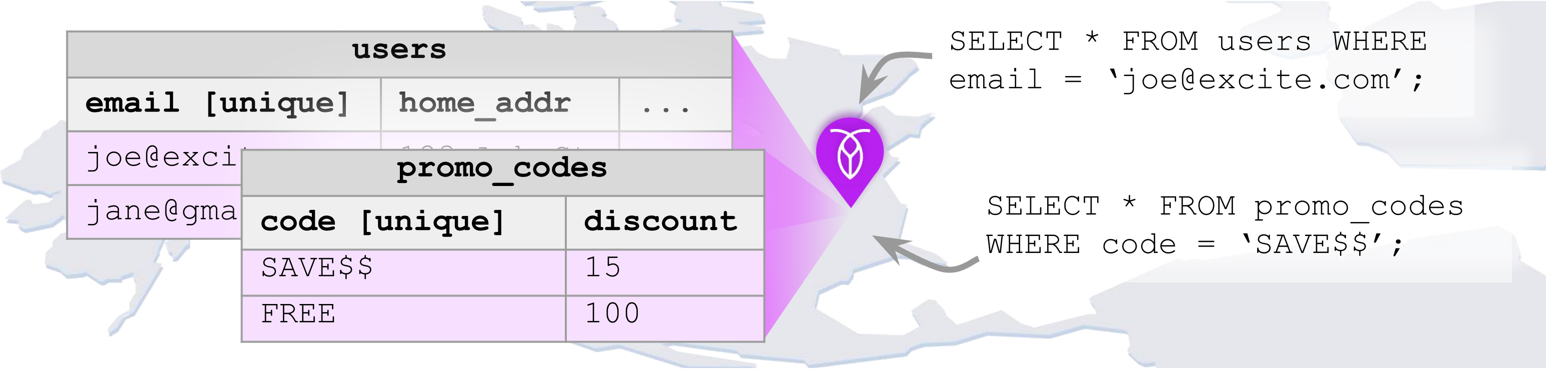
Tables accessed from a single region or amenable to partitioning use locality `REGIONAL`

Read-mostly tables not amenable to partitioning use locality `GLOBAL`

Queries leverage data closest to them



Example: movr application



users

email [unique]	home_addr	...
joe@exci		
jane@gma		

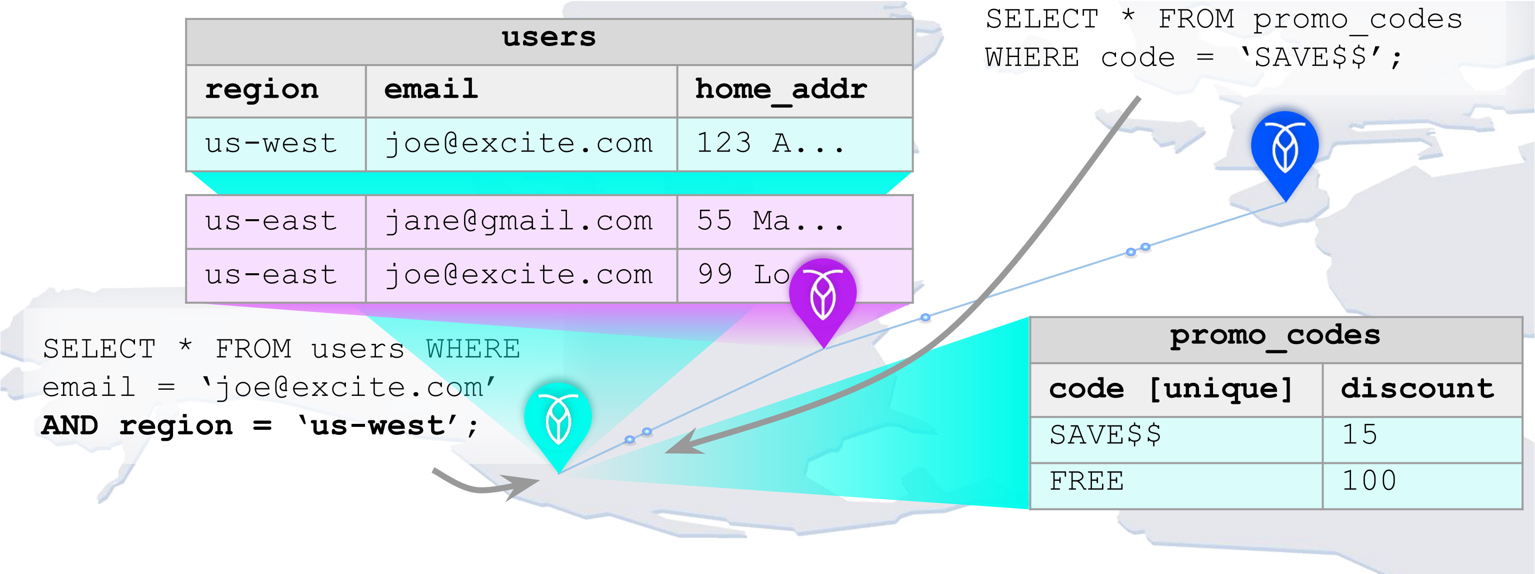
promo_codes

code [unique]	discount
SAVE$$	15
FREE	100

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

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

Example: movr application



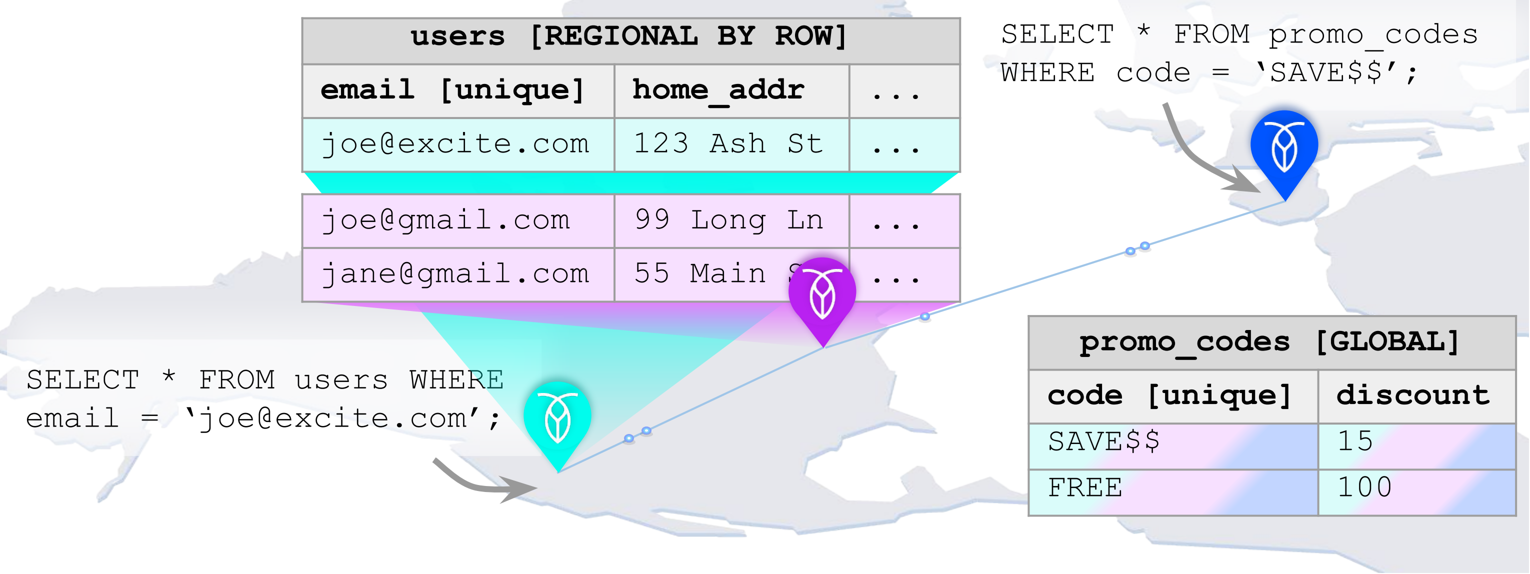
users		
region	**email**	**home_addr**
us-west	joe@excite.com	123 A...
us-east	jane@gmail.com	55 Ma...
us-east	joe@excite.com	99 Lo

```
SELECT * FROM users WHERE
email = 'joe@excite.com'
AND region = 'us-west';
```

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

promo_codes	
code [unique]	**discount**
SAVE$$	15
FREE	100

Example: movr application



users [REGIONAL BY ROW]		
email [unique]	**home_addr**	...
joe@excite.com	123 Ash St	...
joe@gmail.com	99 Long Ln	...
jane@gmail.com	55 Main S	...

```
SELECT * FROM promo_codes
WHERE code = ‘SAVE$$’;
```

```
SELECT * FROM users WHERE
email = ‘joe@excite.com’;
```

promo_codes [GLOBAL]	
code [unique]	**discount**
SAVE$$	15
FREE	100

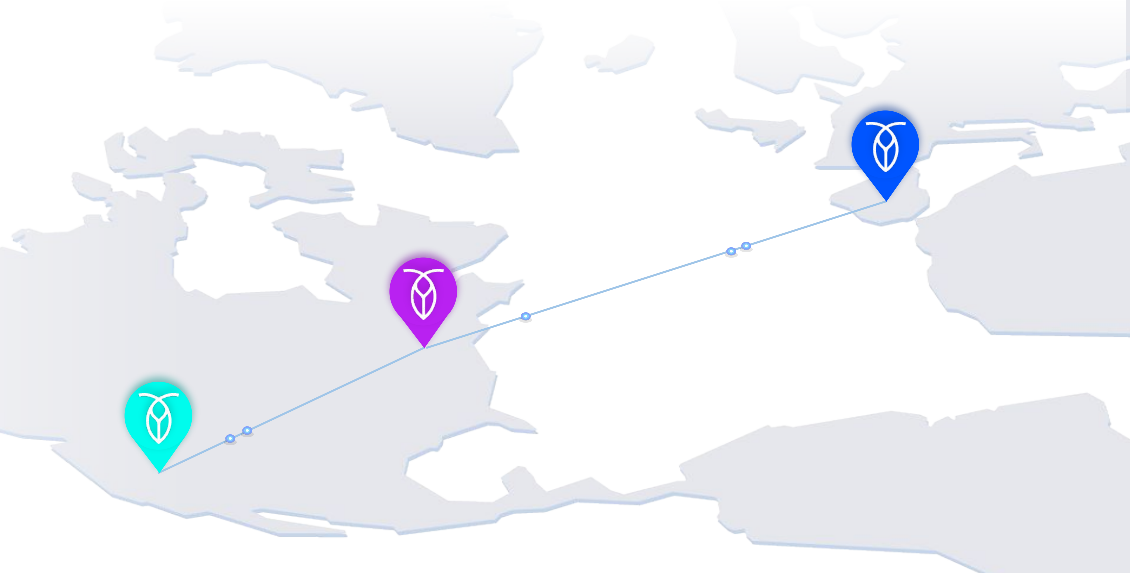
Regional tables

`REGIONAL BY TABLE` v `REGIONAL BY ROW`

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.

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



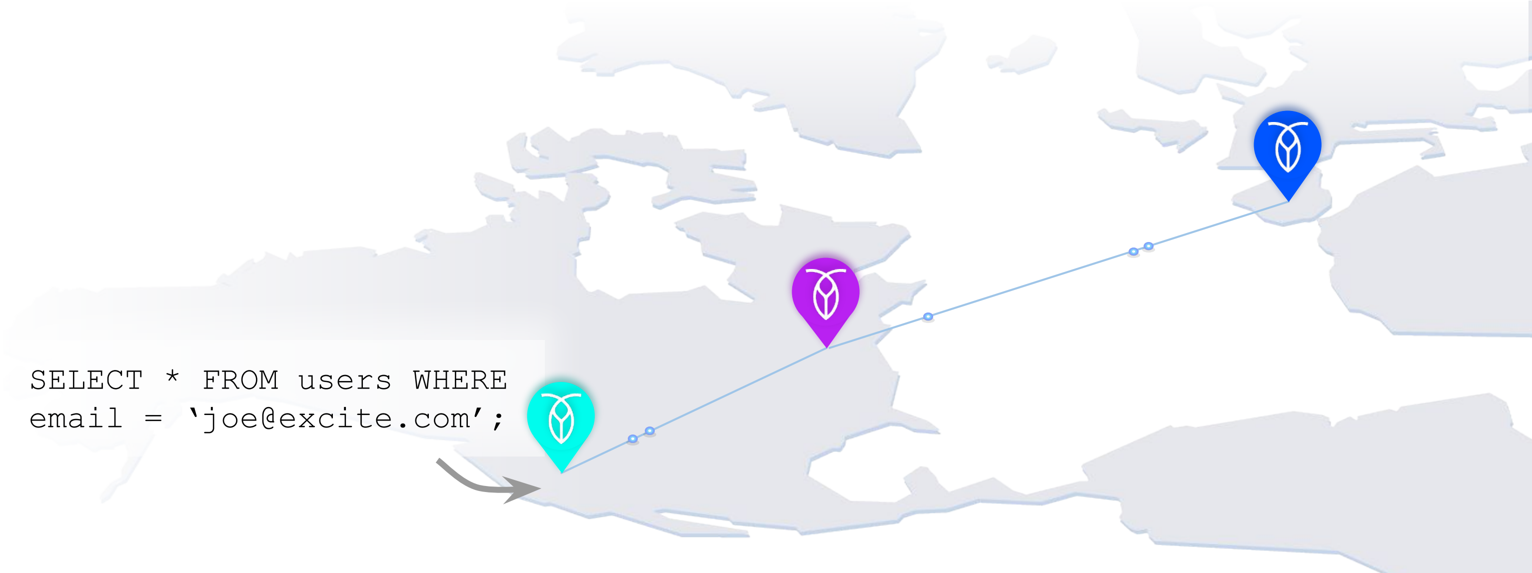
Inserting into a Regional by Row table

```
> 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)
             ├── with-scan &1
             └── 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

```
> 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']
```

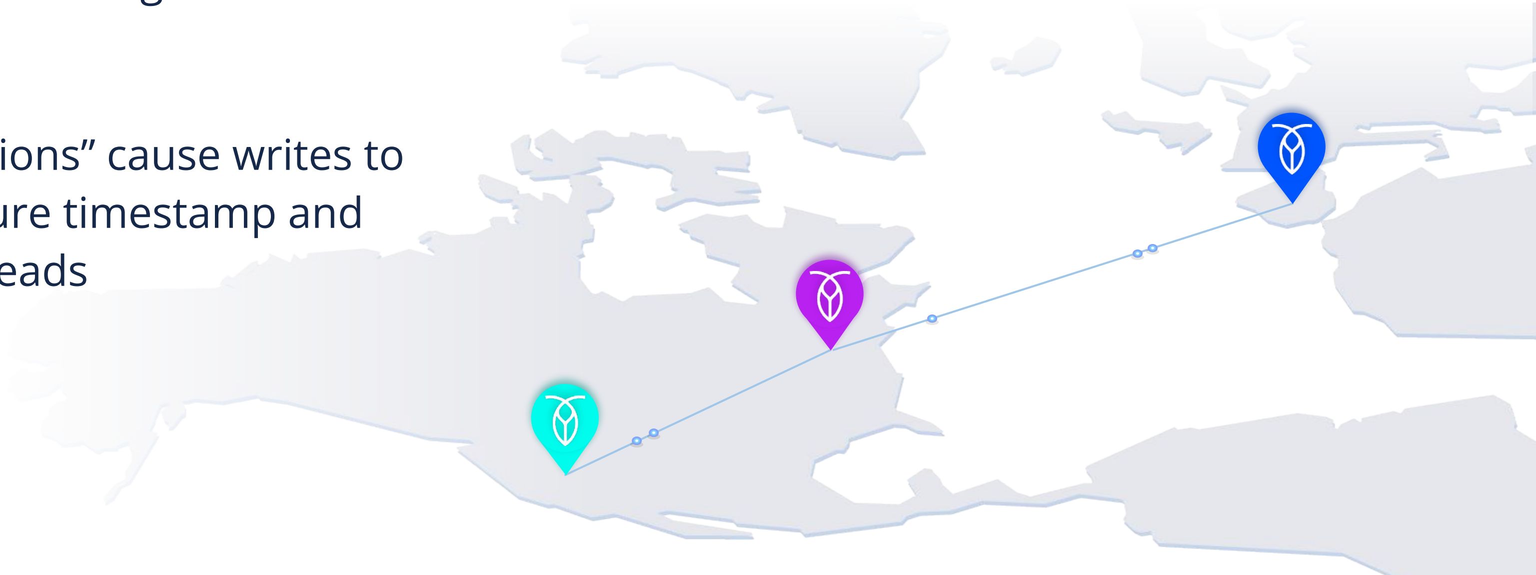
Global tables

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

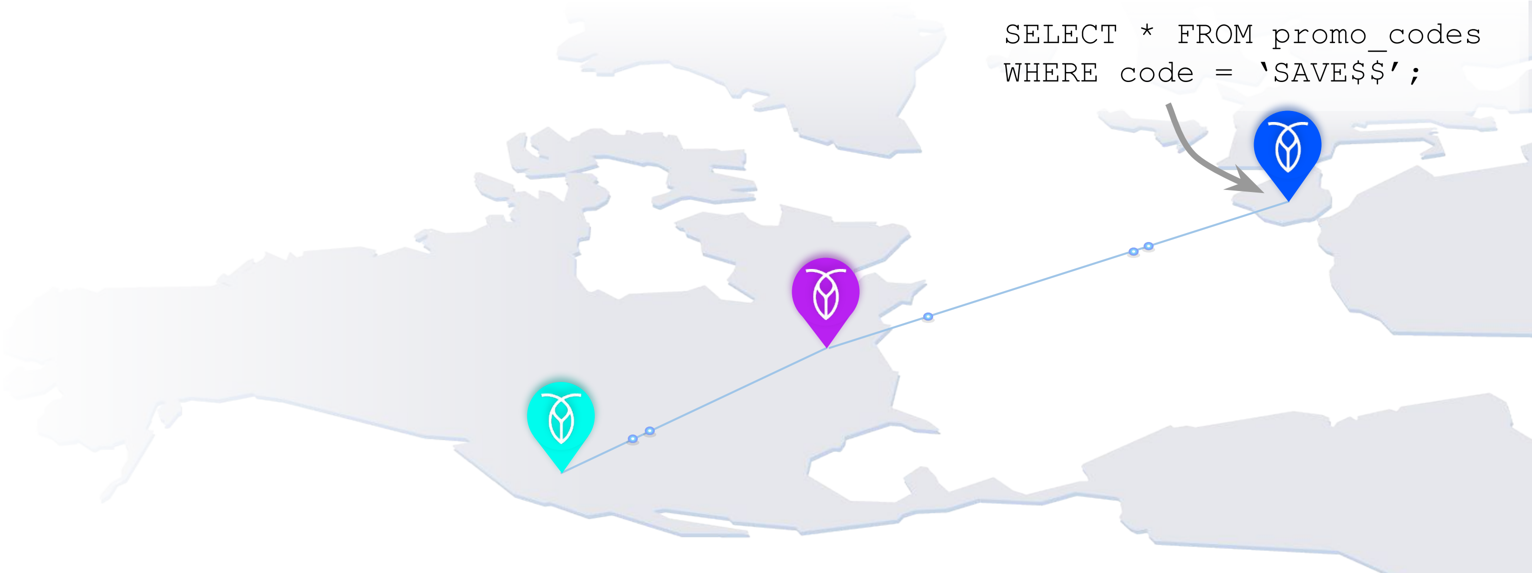
```
CREATE TABLE promo_codes (
    code STRING,
    discount FLOAT
) LOCALITY GLOBAL
```



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

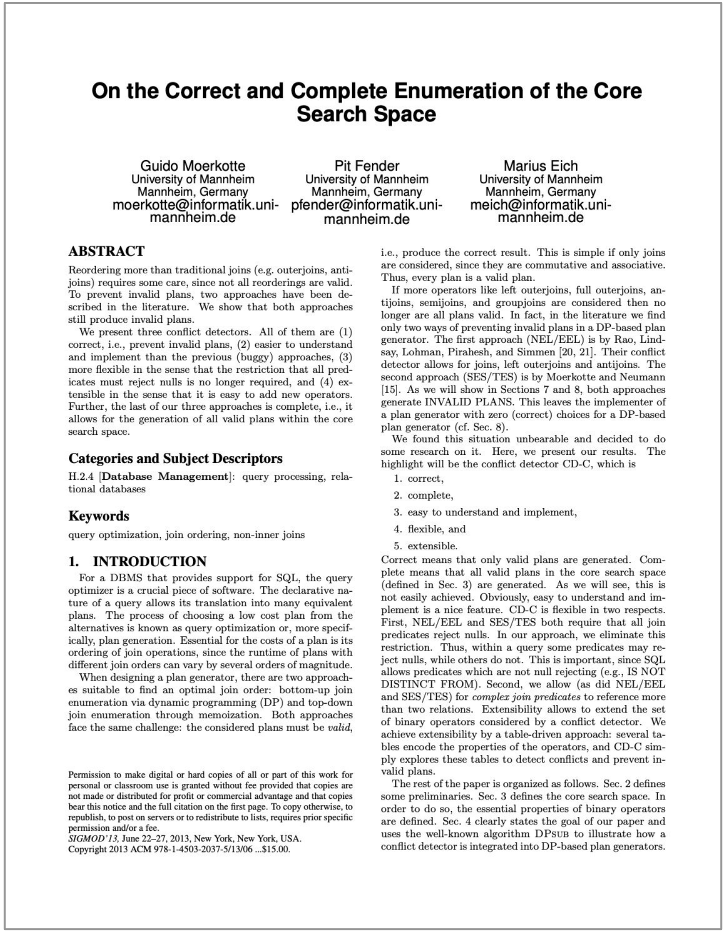
Optimizing for OLTP

- 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



On the Correct and Complete Enumeration of the Core Search Space

Guido Moerkotte
University of Mannheim
Mannheim, Germany
moerkotte@informatik.uni-mannheim.de

Pit Fender
University of Mannheim
Mannheim, Germany
pfender@informatik.uni-mannheim.de

Marius Eich
University of Mannheim
Mannheim, Germany
meich@informatik.uni-mannheim.de

ABSTRACT

Reordering more than traditional joins (e.g. outerjoins, antijoins) requires some care, since not all reorderings are valid. To prevent invalid plans, two approaches have been described in the literature. We show that both approaches still produce invalid plans.

We present three conflict detectors. All of them are (1) correct, i.e., prevent invalid plans, (2) easier to understand and implement than the previous (buggy) approaches, (3) more flexible in the sense that the restriction that all predicates must reject nulls is no longer required, and (4) extensible in the sense that it is easy to add new operators. Further, the last of our three approaches is complete, i.e., it allows for the generation of all valid plans within the core search space.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: query processing, relational databases

Keywords

query optimization, join ordering, non-inner joins

1. INTRODUCTION

For a DBMS that provides support for SQL, the query optimizer is a crucial piece of software. The declarative nature of a query allows its translation into many equivalent plans. The process of choosing a low cost plan from the alternatives is known as query optimization or, more specifically, plan generation. Essential for the costs of a plan is its ordering of join operations, since the runtime of plans with different join orders can vary by several orders of magnitude.

When designing a plan generator, there are two approaches suitable to find an optimal join order: bottom-up join enumeration via dynamic programming (DP) and top-down join enumeration through memoization. Both approaches face the same challenge: the considered plans must be *valid*, i.e., produce the correct result. This is simple if only joins are considered, since they are commutative and associative. Thus, every plan is a valid plan.

If more operators like left outerjoins, full outerjoins, antijoins, semijoins, and groupjoins are considered then no longer are all plans valid. In fact, in the literature we find only two ways of preventing invalid plans in a DP-based plan generator. The first approach (NEL/EEL) is by Rao, Lindsay, Lohman, Pirahesh, and Simmen [20, 21]. Their conflict detector allows for joins, left outerjoins and antijoins. The second approach (SES/TES) is by Moerkotte and Neumann [15]. As we will show in Sections 7 and 8, both approaches generate INVALID PLANS. This leaves the implementer of a plan generator with zero (correct) choices for a DP-based plan generator (cf. Sec. 8).

We found this situation unbearable and decided to do some research on it. Here, we present our results. The highlight will be the conflict detector CD-C, which is

1. correct,
2. complete,
3. easy to understand and implement,
4. flexible, and
5. extensible.

Correct means that only valid plans are generated. Complete means that all valid plans in the core search space (defined in Sec. 3) are generated. As we will see, this is not easily achieved. Obviously, easy to understand and implement is a nice feature. CD-C is flexible in two respects. First, NEL/EEL and SES/TES both require that all join predicates reject nulls. In our approach, we eliminate this restriction. Thus, within a query some predicates may reject nulls, while others do not. This is important, since SQL allows predicates which are not null rejecting (e.g., IS NOT DISTINCT FROM). Second, we allow (as did NEL/EEL and SES/TES) for *complex join predicates* to reference more than two relations. Extensibility allows to extend the set of binary operators considered by a conflict detector. We achieve extensibility by a table-driven approach: several tables encode the properties of the operators, and CD-C simply explores these tables to detect conflicts and prevent invalid plans.

The rest of the paper is organized as follows. Sec. 2 defines some preliminaries. Sec. 3 defines the core search space. In order to do so, the essential properties of binary operators are defined. Sec. 4 clearly states the goal of our paper and uses the well-known algorithm DPsub to illustrate how a conflict detector is integrated into DP-based plan generators.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

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;
  tree | field  | description
+------+--------+-------------+
  scan |        |
       | table  | ab@b
       | spans  | ALL
       | filter | 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;

    tree    |       field        | description
+-----------+--------------------+-------------+
  hash-join |                    |
   │        | type               | inner
   │        | equality           | (a) = (c)
   │        | left cols are key  |
   │        | right cols are key |
   ├── scan |                    |
   │        | table              | ab@primary
   │        | spans              | ALL
   └── scan |                    |
            | table              | cd@primary
            | spans              | ALL
```

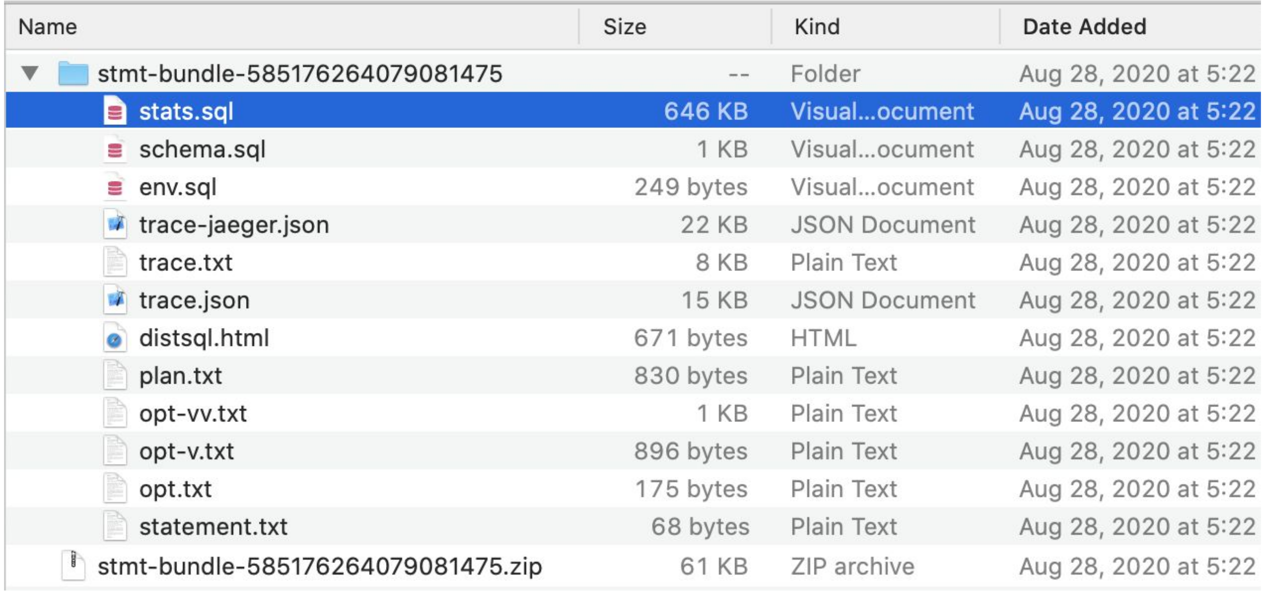
Debugging tools

```
EXPLAIN ANALYZE (DEBUG) SELECT …
                                     text
--------------------------------------------------------------------------------
  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 …
```



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

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