CockroachDB

Architecture of a Geo-Distributed SQL Database

Peter Mattis (@petermattis), Co-founder & CTO
CockroachDB: Geo-distributed SQL Database

Make Data Easy

• Distributed
  ○ Horizontally scalable to grow with your application
• Geo-distributed
  ○ Handle datacenter failures
  ○ Place data near usage
  ○ Push computation near data
• SQL
  ○ Lingua-franca for rich data storage
  ○ Schemas, indexes, and transactions make app development easier
AGENDA

- Introduction
- Ranges and Replicas
- Transactions
- SQL Data in a KV World
- SQL Execution
- SQL Optimization
Distributed, Replicated, Transactional KV*

• Keys and values are strings
  o Lexicographically ordered by key
• Multi-version concurrency control (MVCC)
  o Values are never updated “in place”, newer versions shadow older versions
  o Tombstones are used to delete values
  o Provides snapshot to each transaction
• Monolithic key-space

* Not exposed for external usage

Cockroach Labs
Monolithic Key Space

Monolithic logical key space

- Ordered lexicographically by key
Ranges

Key space divided into contiguous ~64MB ranges

Ranges are small enough to be moved/split quickly

Ranges are large enough to amortize indexing overhead
Range Indexing

Index structure used to locate ranges (very much like a B-tree)
Ordered Range Scans

Ordered keys enable efficient range scans

dogs >= “muddy” AND <= “stella”
Transactional Updates

Transactions used to insert records into ranges

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>carl - jack</td>
</tr>
<tr>
<td>2</td>
<td>lady - peetey</td>
</tr>
<tr>
<td>3</td>
<td>pinetop - zee</td>
</tr>
</tbody>
</table>

Space available in range? - YES
Transactional Updates

Transactions used to insert records into ranges

**DOGS**
carl  
dagne  
figment  
jack  
lady  
lula  
muddy  
peetey  
pinetop  
sooshi  
stella  
zee

1. carl - jack  
2. lady - peetey  
3. pinetop - zee

- carl  
- dagne  
- figment  
- jack  
- lady  
- lula  
- muddy  
- peetey  
- pinetop  
- sooshi  
- stella  
- sunny  
- zee

**INSERT[sunny]**
Range Splits

**DOGS**
- carl
- dagne
- figment
- jack
- lady
- lula
- muddy
- peetey
- pinetop
- sooshi
- stella
- sunny
- zee

**INSERT [rudy]**

**1** carl - jack
**2** lady - peetey
**3** pinetop - zee

**BUT...** what happens when a range is full?

**SPACE AVAILABLE IN RANGE? - NO**
Range Splits

Ranges are automatically split, a new range index is created & order maintained

1 carl - jack
2 lady - peetey
3 pinetop - sooshi
4 stella - zee

- carl
- dagne
- figment
- jack
- lady
- lula
- muddy
- peetey
- pinetop
- sooshi
- stella
- zee

INSERT [rudy]

split range and insert
Raft and Replication

Ranges (~64MB) are the unit of replication

Each range is a Raft group
(Raft is a consensus replication protocol)

Default to 3 replicas, though this is configurable

- Important system ranges default to 5 replicas
- Note: 2 replicas doesn’t make sense in consensus replication
Raft and Replication

Raft provides “atomic replication” of commands

Commands are proposed by the leaseholder replica and distributed to the follower replicas, but only accepted when a quorum of replicas have acknowledged receipt

* Leaseholder == Raft leader
Range Leases

Reads with consensus
Reads must talk to a quorum of replicas
Range Leases

Reads without consensus
One replica is chosen as the leaseholder
Reads without consensus
One replica is chosen as the leaseholder
- Coordinates writes (proposal, key locking)
- Performs reads
Replica Placement

- Space
- Diversity
- Load
- Latency

Each Range is a Raft state machine
A Range has 1 or more Replicas
Replica Placement: Diversity

Diversity optimizes placement of replicas across “failure domains”

- Disk
- Single machine
- Rack
- Datacenter
- Region
Replica Placement: Load

Load
Balances placement using heuristics that considers real-time usage metrics of the data itself.

This range is high load as it is accessed more than others.

While we show this for ranges within a single table, this is also applicable across all ranges across ALL tables, which is the more typical situation.
We apply a constraint that indicates regional placement so we can ensure low latency access or jurisdictional control of data.
Rebalancing Replicas

Scale: Add a node
If we add a node to the cluster, CockroachDB automatically redistributed replicas to even load across the cluster

Uses the replica placement heuristics from previous slides to decide which node to add to and which to remove from...
Rebalancing Replicas

**Scale: Add a node**
If we add a node to the cluster, CockroachDB automatically redistributed replicas to even load across the cluster.

Uses the replica placement heuristics from previous slides.

Movement is decomposed into adding a replica followed by removing a replica.
Rebalancing Replicas

Scale: Add a node
If we add a node to the cluster, CockroachDB automatically redistributed replicas to even load across the cluster.

Uses the replica placement heuristics from previous slides.

Movement is decomposed into adding a replica followed by removing a replica.
Rebalancing Replicas

Loss of a node

**Permanent Failure**

If a node goes down, the Raft group realizes a replica is missing and replaces it with a new replica on an active node.

Uses the replica placement heuristics from previous slides.
Rebalancing Replicas

Loss of a node

**Permanent Failure**

If a node goes down, the Raft group realizes a replica is missing and replaces it with a new replica on an active node.

Uses the replica placement heuristics from previous slides.

The failed replica is removed from the Raft group and a new replica created. The leaseholder sends a snapshot of the Range’s state to bring the new replica up to date.
Rebalancing Replicas

Loss of a node

Temporary Failure
If a node goes down for a moment, the leaseholder can “catch up” any replica that is behind.

The leaseholder can send commands to be replayed OR it can send a snapshot of the current Range data. We apply heuristics to decide which is most efficient for a given failure.
AGENDA

- Introduction
- Ranges and Replicas
- Transactions
- SQL Data in a KV World
- SQL Execution
- SQL Optimization
Transactions

Atomicity, Consistency, Isolation, Durability

Serializable Isolation

- As if the transactions are run in a serial order
- Gold standard isolation level
- Make Data Easy - weaker isolation levels are too great a burden

Transactions can span arbitrary ranges

Conversational

- The full set of operations is not required up front
Transactions

Raft provides atomic writes to individual ranges

Bootstrap transaction atomicity using Raft atomic writes

Transaction record atomically flipped from PENDING to COMMIT
Distributed Transactions

```
INSERT INTO dogs
VALUES (sunny, ozzie)
```
Distributed Transactions

```
BEGIN TXN1
WRITE[sunny]
```

```
INSERT INTO dogs
VALUES (sunny, ozzie)
```
Distributed Transactions

BEGIN TXN1
WRITE[sunny]

transactions
TXN1: PENDING

INSERT INTO dogs
VALUES (sunny, ozzie)
Distributed Transactions

BEGIN TXN1
WRITE[sunny]

INSERT INTO dogs VALUES (sunny, ozzie)

transactions
TXN1: PENDING

ACK

node 1
node 2
node 3
node 4

carl
dagne
figment
jack
lady
lula
muddy
peetey

pinetop
sooshi
stella
sunny
zee

carl
dagne
figment
jack

lady
lula
muddy
peetey

pinetop
sooshi
stella
sunny
zee

carl
dagne
figment
jack

lady
lula
muddy
peetey

carl
dagne
figment
jack

lady
lula
muddy
peetey

carl
dagne
figment
jack

lady
lula
muddy
peetey

LADY
BEGIN TXN1
WRITE[sunny]

INSERT INTO dogs VALUES (sunny, ozzie)

transactions TXN1: PENDING
Distributed Transactions

BEGIN TXN1
WRITE[sunny]
WRITE[ozzie]

transactions
TXN1: PENDING

ACK

INSERT INTO dogs
VALUES (sunny, ozzie)
Distributed Transactions

BEGIN TXN1
WRITE[sunny]
WRITE[oizzie]

transactions

TXN1: PENDING
Distributed Transactions

BEGIN TXN1
WRITE[sunny]
WRITE[ozzie]

GATEWAY

transactions
TXN1: PENDING

INSERT INTO dogs VALUES (sunny, ozzie)
Distributed Transactions

BEGIN TXN1
WRITE[sunny]
WRITE[ozzie]

INSERT INTO dogs
VALUES (sunny, ozzie)
Distributed Transactions

BEGIN TXN1
WRITE[sunny]
WRITE[ozzie]
COMMIT

GATEWAY

INSERT INTO dogs
VALUES (sunny, ozzie)

transactions
TXN1: COMMIT
Distributed Transactions

BEGIN TXN1
WRITE[sunny]
WRITE[ozzie]
COMMIT

GATEWAY

INSERT INTO dogs
VALUES (sunny, ozzie)
Transactions: Pipelining
Transactions: Pipelining

BEGIN
WRITE[sunny]

Serial

Pipelined

txn:sunny (pending)
sunny

sunny
Transactions: Pipelining

Serial

Pipelined

BEGIN
WRITE[sunny]
WRITE[ozzie]
Transactions: Pipelining

BEGIN
WRITE[sunny]
WRITE[ozzie]
COMMIT
Transactions: Pipelining

BEGIN
WRITE[sunny]
WRITE[ozzie]
COMMIT

Committed once all operations complete

We replaced the centralized commit marker with a distributed one

* "Proved" with TLA+
AGENDA

- Introduction
- Ranges and Replicas
- Transactions
- SQL Data in a KV World
- SQL Execution
- SQL Optimization
SQL

Structured Query Language

Declarative, not imperative
• These are the results I want vs perform these operations in this sequence

Relational data model
• Typed: INT, FLOAT, STRING, ...
• Schemas: tables, rows, columns, foreign keys
SQL: Tabular Data in a KV World

SQL data has columns and types?!?

How do we store typed and columnar data in a distributed, replicated, transactional key-value store?
  • The SQL data model needs to be mapped to KV data
  • Reminder: keys and values are lexicographically sorted
SQL Data Mapping: Inventory Table

CREATE TABLE inventory (  
id INT PRIMARY KEY,  
name STRING,  
price FLOAT  
)

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bat</td>
<td>1.11</td>
</tr>
<tr>
<td>2</td>
<td>Ball</td>
<td>2.22</td>
</tr>
<tr>
<td>3</td>
<td>Glove</td>
<td>3.33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>/1</td>
<td>“Bat”,1.11</td>
</tr>
<tr>
<td>/2</td>
<td>“Ball”,2.22</td>
</tr>
<tr>
<td>/3</td>
<td>“Glove”,3.33</td>
</tr>
</tbody>
</table>
### SQL Data Mapping: Inventory Table

CREATE TABLE inventory (  
id INT PRIMARY KEY,  
name STRING,  
price FLOAT  
)

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bat</td>
<td>1.11</td>
</tr>
<tr>
<td>2</td>
<td>Ball</td>
<td>2.22</td>
</tr>
<tr>
<td>3</td>
<td>Glove</td>
<td>3.33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>/&lt;Table&gt;//&lt;Index&gt;/1</td>
<td>“Bat”, 1.11</td>
</tr>
<tr>
<td>/&lt;Table&gt;//&lt;Index&gt;/2</td>
<td>“Ball”, 2.22</td>
</tr>
<tr>
<td>/&lt;Table&gt;//&lt;Index&gt;/3</td>
<td>“Glove”, 3.33</td>
</tr>
</tbody>
</table>
SQL Data Mapping: Inventory Table

CREATE TABLE inventory (  
id INT PRIMARY KEY,  
name STRING,  
price FLOAT  
)
CREATE TABLE inventory (  
id INT PRIMARY KEY,  
name STRING,  
price FLOAT,  
INDEX name_idx (name)  
)
CREATE TABLE inventory (  
id INT PRIMARY KEY,  
name STRING,  
price FLOAT,  
INDEX name_idx (name)  
)

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bat</td>
<td>1.11</td>
</tr>
<tr>
<td>2</td>
<td>Ball</td>
<td>2.22</td>
</tr>
<tr>
<td>3</td>
<td>Glove</td>
<td>3.33</td>
</tr>
<tr>
<td>4</td>
<td>Bat</td>
<td>4.44</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>/inventory/name_idx/&quot;Bat&quot;/1</td>
<td>∅</td>
</tr>
<tr>
<td>/inventory/name_idx/&quot;Ball&quot;/2</td>
<td>∅</td>
</tr>
<tr>
<td>/inventory/name_idx/&quot;Glove&quot;/3</td>
<td>∅</td>
</tr>
</tbody>
</table>
CREATE TABLE inventory (  id INT PRIMARY KEY,  name STRING,  price FLOAT,  INDEX name_idx (name) )

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bat</td>
<td>1.11</td>
</tr>
<tr>
<td>2</td>
<td>Ball</td>
<td>2.22</td>
</tr>
<tr>
<td>3</td>
<td>Glove</td>
<td>3.33</td>
</tr>
<tr>
<td>4</td>
<td>Bat</td>
<td>4.44</td>
</tr>
</tbody>
</table>

Key | Value
--- | ---
/inventory/name_idx/"Bat"/1 | Ø
/inventory/name_idx/"Ball"/2 | Ø
/inventory/name_idx/"Glove"/3 | Ø
/inventory/name_idx/"Bat"/4 | Ø
AGENDA

- Introduction
- Ranges and Replicas
- Transactions
- SQL Data in a KV World
- SQL Execution
- SQL Optimization
SQL Execution

Relational operators

- Projection (SELECT <columns>)
- Selection (WHERE <filter>)
- Aggregation (GROUP BY <columns>)
- Join (JOIN), union (UNION), intersect (INTERSECT)
- Scan (FROM <table>)
- Sort (ORDER BY)
  - Technically, not a relational operator
SQL Execution

• Relational expressions have input expressions and scalar expressions
  ○ For example, a “filter” expression has 1 input expression and a scalar expression that filters the rows from the child
  ○ The scan expression has zero inputs

• Query plan is a tree of relational expressions
• SQL execution takes a query plan and runs the operations to completion
SQL Execution: Example

SELECT name
FROM   inventory
WHERE  name >= "b" AND name < "c"
SQL Execution: Scan

SELECT name
FROM inventory
WHERE name >= "b" AND name < "c"
SQL Execution: Filter

```
SELECT name
FROM   inventory
WHERE  name >= "b" AND name < "c"
```
SELECT name
FROM inventory
WHERE name >= "b" AND name < "c"
SQL Execution: Project

SELECT name
FROM inventory
WHERE name >= "b" AND name < "c"

Scan inventory → Filter name >= “b” AND name < “c” → Project name → Results
SQL Execution: Index Scans

```
SELECT name
FROM inventory
WHERE name >= "b" AND name < "c"
```

Scan
inventory@name ["b" - "c")

The filter gets pushed into the scan
SQL Execution: Index Scans

```sql
SELECT name
FROM   inventory
WHERE  name >= "b" AND name < "c"
```
SQL Execution: Correctness

Correct SQL execution involves lots of bookkeeping
- User defined tables, and indexes
- Queries refer to table and column names
- Execution uses table and column IDs
- NULL handling
SQL Execution: Performance

Performant SQL execution
• Tight, well written code
• Operator specialization
  ○ hash group by, stream group by
  ○ hash join, merge join, lookup join, zig-zag join
• Distributed execution
SQL Execution: Group By

```
SELECT COUNT(*), country
FROM customers
GROUP BY country
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>
### SQL Execution: Hash Group By

```
SELECT COUNT(*), country
FROM customers
GROUP BY country
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>
SQL Execution: Hash Group By

```sql
SELECT COUNT(*), country 
FROM customers 
GROUP BY country
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>
### SQL Execution: Hash Group By

```sql
SELECT COUNT(*), country
FROM customers
GROUP BY country
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>United States</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Germany</td>
<td>1</td>
</tr>
</tbody>
</table>
SQL Execution: Hash Group By

```sql
SELECT COUNT(*), country
FROM customers
GROUP BY country
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>
SQL Execution: Hash Group By

```
SELECT COUNT(*), country
FROM customers
GROUP BY country
```

<table>
<thead>
<tr>
<th>United States</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Germany</td>
<td>1</td>
</tr>
<tr>
<td>France</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>
SQL Execution: Hash Group By

SELECT COUNT(*), country
FROM customers
GROUP BY country

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>United States</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Germany</td>
<td>1</td>
</tr>
<tr>
<td>France</td>
<td>2</td>
</tr>
</tbody>
</table>
SELECT COUNT(*), country
FROM customers
GROUP BY country

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>
SQL Execution: Sort on Grouping Column(s)

```
SELECT COUNT(*), country
FROM customers
GROUP BY country
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>
SQL Execution: Streaming Group By

```sql
SELECT COUNT(*), country
FROM customers
GROUP BY country
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>
### SQL Execution: Streaming Group By

```sql
SELECT COUNT(*), country
FROM customers
GROUP BY country
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>
SQL Execution: Streaming Group By

```sql
SELECT COUNT(*), country
FROM customers
GROUP BY country
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>
## SQL Execution: Streaming Group By

```sql
SELECT   COUNT(*), country
FROM     customers
GROUP BY country
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>
SQL Execution: Streaming Group By

```
SELECT COUNT(*), country 
FROM customers 
GROUP BY country 
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacques</td>
<td>France</td>
</tr>
<tr>
<td>Marie</td>
<td>France</td>
</tr>
<tr>
<td>Hans</td>
<td>Germany</td>
</tr>
<tr>
<td>Bob</td>
<td>United States</td>
</tr>
<tr>
<td>Susan</td>
<td>United States</td>
</tr>
</tbody>
</table>
Distributed SQL Execution

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

Push fragments of computation as close to the data as possible.
Distributed SQL Execution: Streaming Group By

```
SELECT COUNT(*), country
FROM customers
GROUP BY country
```
Distributed SQL Execution: Streaming Group By

SELECT COUNT(*), country
FROM customers
GROUP BY country
Distributed SQL Execution: Streaming Group By

```
SELECT COUNT(*), country
FROM customers
GROUP BY country
```
AGENDA

- Introduction
- Ranges and Replicas
- Transactions
- SQL Data in a KV World
- SQL Execution
- SQL Optimization
SQL Optimization

An optimizer explores many plans that are logically equivalent to a given query and chooses the best one.

Parse SQL
- Parse SQL

Prep
- Fold Constants
- Check Types
- Resolve Names
- Report Semantic Errors
- Compute properties
- Retrieve and attach stats
- Cost-independent transformations

Search
- Cost-based transformations

Execute
- Plan

Diagram:
- Parse -> Prep -> Search -> Execute
- Parse SQL
- AST
- Memo
- Plan
- Execute
SQL Optimization: Cost-Independent Transformations

- Some transformations always make sense
  - Constant folding
  - Filter push-down
  - Decorrelating subqueries*
  - ...

- These transformations are cost-independent
  - If the transformation can be applied to the query, it is applied

- **Domain Specific Language** for transformations
  - Compiled down to code which efficiently matches query fragments in the memo
  - ~200 transformations currently defined

* Actually cost-based, but we’re treating it as cost-independent right now
SQL Optimization: Filter Push-Down

SELECT * FROM a JOIN b WHERE x > 10

Initial plan

Scan a@primary
Scan b@primary
Join
Filter x > 10
Results
SQL Optimization: Filter Push-Down

```
SELECT * FROM a JOIN b WHERE x > 10
```

After filter push-down: Scan a@primary, Filter x > 10, Scan b@primary, Filter x > 10, Join, Results
SQL Optimization: Cost-Based Transformations

• Some transformations are not universally good
  ○ Index selection
  ○ Join reordering
  ○ ...

• These transformations are cost-based
  ○ When should the transformation be applied?
  ○ Need to try both paths and maintain both the original and transformed query
  ○ State explosion: thousands of possible query plans
    ■ Memo data structure maintains a forest of query plans
  ○ Estimate cost of each query, select query with lowest cost

• Costing
  ○ Based on table statistics and estimating cardinality of inputs to relational expressions
SQL Optimization: Cost-based Index Selection

The index to use for a query is affected by multiple factors
- Filters and join conditions
- Required ordering (ORDER BY)
- Implicit ordering (GROUP BY)
- Covering vs non-covering (i.e. is an index-join required)
- Locality
SQL Optimization: Cost-based Index Selection

```sql
SELECT * FROM a WHERE x > 10 ORDER BY y
```

Required orderings affect index selection
Sorting is expensive if there are a lot of rows
Sorting can be the better option if there are few rows
SQL Optimization: Cost-based Index Selection

```sql
SELECT * FROM a WHERE x > 10 ORDER BY y
```

Required orderings affect index selection
- Sorting is expensive if there are a lot of rows
- Sorting can be the better option if there are few rows
SQL Optimization: Cost-based Index Selection

Required orderings affect index selection

Sorting is expensive if there are a lot of rows

Sorting can be the better option if there are few rows

SELECT *
FROM a
WHERE x > 10
ORDER BY y
SQL Optimization: Cost-based Index Selection

SELECT * FROM a WHERE x > 10
ORDER BY y

Required orderings affect index selection
Sorting is expensive if there are a lot of rows
Sorting can be the better option if there are few rows
SQL Optimization: Cost-based Index Selection

Required orderings affect index selection

Sorting is expensive if there are a lot of rows

Sorting can be the better option if there are few rows

```sql
SELECT   *
FROM     a
WHERE    x > 10
ORDER BY y
```
SQL Optimization: Cost-based Index Selection

Required orderings affect index selection

Sorting is expensive if there are a lot of rows
Sorting can be the better option if there are few rows

SELECT * FROM a WHERE x > 10 ORDER BY y

Scan a@primary
Filter x > 10
Sort y

Scan a@x [10 - )
Sort y
50,000

Scan a@y
Filter x > 10
50,000

100,000

50,000

50,000

Lowest Cost
Locality-Aware SQL Optimization

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

Duplicate read-mostly data in each locality.

Plan queries to use data from the same locality.
Locality-Aware SQL Optimization

Three copies of the `postal_codes` table data

Use replication constraints to pin the copies to different geographic regions (US-East, US-West, EU)

```sql
CREATE TABLE postal_codes (
    id INT PRIMARY KEY,
    code STRING,
    INDEX idx_eu (id) STORING (code),
    INDEX idx_usw (id) STORING (code)
)
```
Locality-Aware SQL Optimization

Optimizer includes locality in cost model

Automatically selects index from same locality: primary, idx_eu, or idx_usw

CREATE TABLE postal_codes (
    id INT PRIMARY KEY,
    code STRING,
    INDEX idx_eu (id) STORING (code),
    INDEX idx_usw (id) STORING (code)
)
Conclusion

- Distributed, replicated, transactional key-value store
- Monolithic key space
- Raft replication of ranges (~64MB)
- Replica placement signals: space, diversity, load, latency
- Pipelined transaction operations
- Mapping SQL data to KV storage
- Distributed SQL execution
- Distributed SQL optimization
Thank You

www.cockroachlabs.com

github.com/cockroachdb/cockroach
A Simple Transaction

INSERT INTO DOGS (sunny);
A Simple Transaction: One Range

NOTE: a gateway can be ANY CockroachDB instance. It can find the leaseholder for any range and execute a transaction
A Simple Transaction: One Range

BEGIN
WRITE[sunny]
COMMIT

GATEWAY

INSERT INTO DOGS (sunny);
A Simple Transaction: One Range

```
BEGIN
WRITE[sunny]
COMMIT
```

```
INSERT INTO DOGS (sunny);
```
Ranges

CockroachDB implements order-preserving data distribution

- Automates sharding of key/value data into “ranges”
- Supports efficient range scans
- Requires an indexing structure

Foundational capability that enables efficient distribution of data across nodes within a CockroachDB cluster

* This approach is also used by Bigtable (tablets), HBase (regions) & Spanner (ranges)