Spanner SQL

CSE 532 · Database Systems

November 4, 2026

Bacon, Bales, Bruno, Cooper, Dickinson, Fikes, Fraser, Gubarev, Joshi, Kogan, Lloyd, Melnik, Rao, Shue, Taylor, van der Holst, and Woodford (Google)

Spanner: Becoming a SQL System · SIGMOD 2017

Why Spanner became a SQL system

- Spanner began as a key-value store: point lookups, range scans, multi-row transactions, reads at a timestamp.

Why Spanner became a SQL system

- Spanner began as a key-value store: point lookups, range scans, multi-row transactions, reads at a timestamp.
- On Bigtable, Google's earlier key-value store, developers lacked a schema, cross-row transactions and a query language, so their code did its own joins and aggregations.

Why Spanner became a SQL system

- Spanner began as a key-value store: point lookups, range scans, multi-row transactions, reads at a timestamp.
- On Bigtable, Google's earlier key-value store, developers lacked a schema, cross-row transactions and a query language, so their code did its own joins and aggregations.
- Wanted: one system for transactions (OLTP), in milliseconds, and analytics, in seconds to hours.

Turn a scalable key-value store into a full SQL database.

The first query engine was bolted on: it used high-level APIs, like an external application.

From one machine to many

	System R, last time	Spanner SQL, today
Data	one machine’s disk	
Cost	page fetches	
Plan	fixed at compile time	
Failure	the query fails	

From one machine to many

	System R, last time	Spanner SQL, today
Data	one machine’s disk	key ranges (**shards**) across datacenters
Cost	page fetches	
Plan	fixed at compile time	
Failure	the query fails	

From one machine to many

	System R, last time	Spanner SQL, today
Data	one machine’s disk	key ranges (**shards**) across datacenters
Cost	page fetches	calls between machines, rows shipped
Plan	fixed at compile time	
Failure	the query fails	

From one machine to many

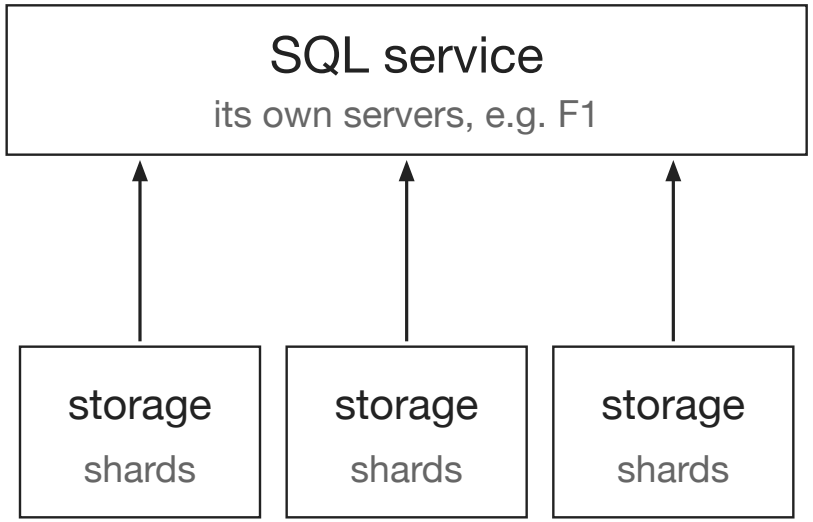
	System R, last time	Spanner SQL, today
Data	one machine's disk	key ranges (**shards**) across datacenters
Cost	page fetches	calls between machines, rows shipped
Plan	fixed at compile time	which shards, and where: at run time
Failure	the query fails	

From one machine to many

	System R, last time	Spanner SQL, today
Data	one machine's disk	key ranges (**shards**) across datacenters
Cost	page fetches	calls between machines, rows shipped
Plan	fixed at compile time	which shards, and where: at run time
Failure	the query fails	a query at one timestamp resumes

Two ways to add SQL

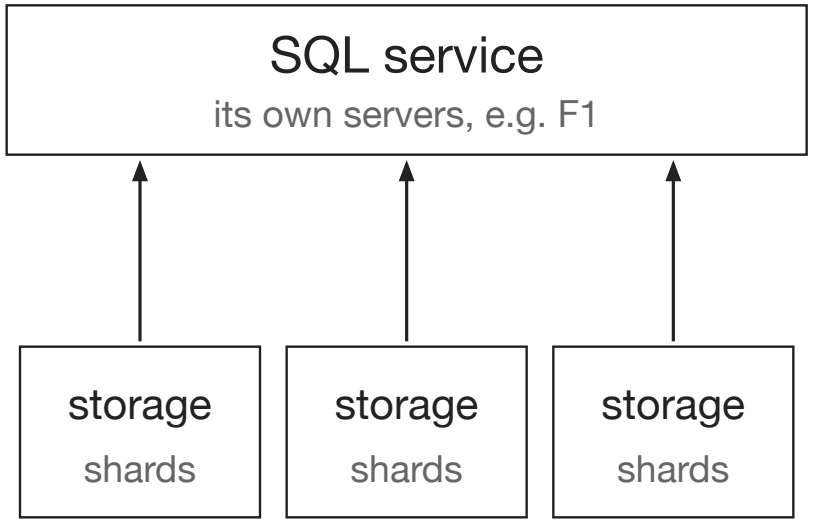
Loosely coupled: SQL beside the store



The SQL service reads key ranges; rows come up.

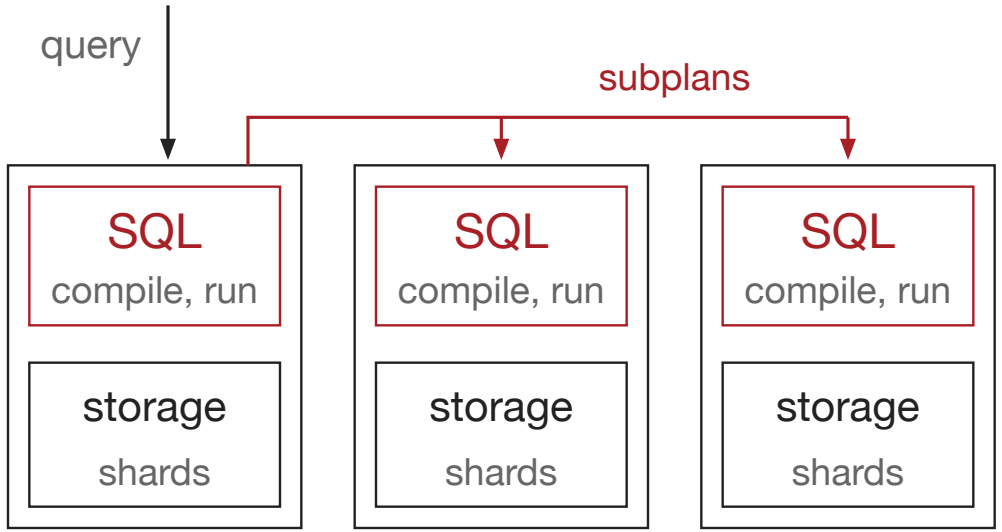
Two ways to add SQL

Loosely coupled: SQL beside the store



The SQL service reads key ranges; rows come up.

Tightly coupled: SQL inside the store



The servers that hold the shards run the plan.

Spanner SQL is the tight design: SQL that knows where the shards are.

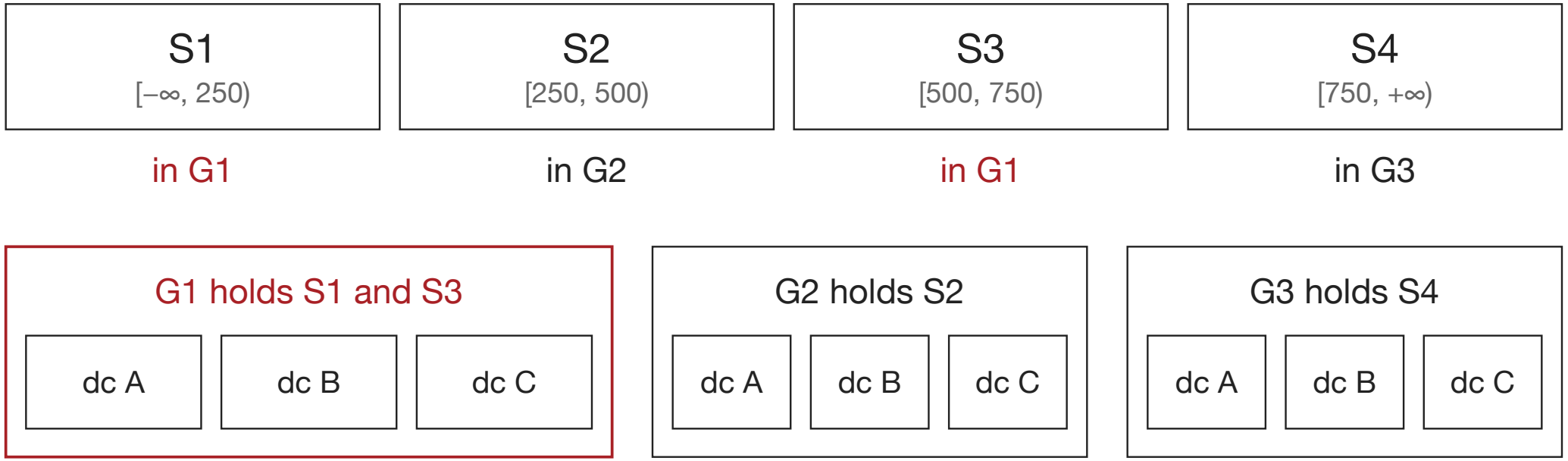
TiDB (Aug 31) sits in between: its SQL processes ship single-table plan pieces into its store, TiKV.

Revisit Spanner

Shards and Paxos groups

Rows sorted by key are cut into key ranges, **shards**; each lives in one **Paxos group**.

Customer, in ckey order, cut into four shards



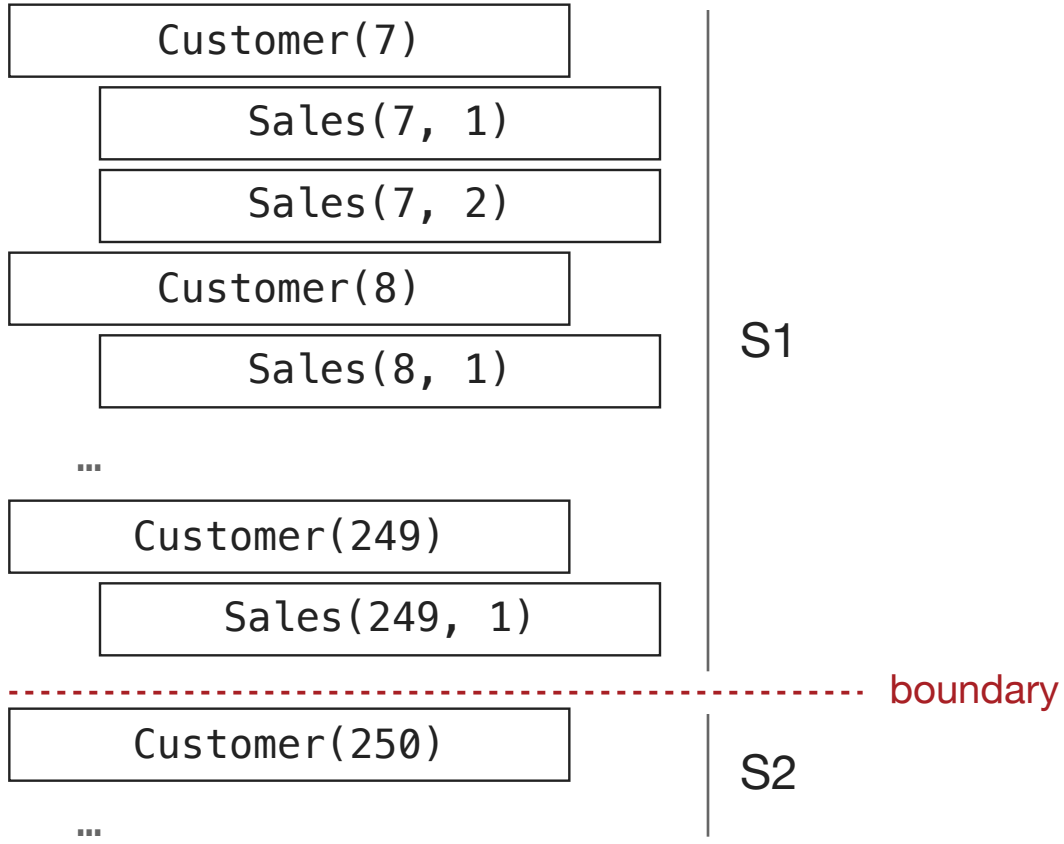
Each Paxos group: one replica in each of three datacenters, A, B and C.

A shard is a unit of key range; a Paxos group, a unit of replication. Every example uses this map.

Interleaving: a child inside its parent

```
CREATE TABLE Customer (
  ckey INT64, name STRING, city STRING
) PRIMARY KEY (ckey)
CREATE TABLE Sales (
  ckey INT64, skey INT64,
  type STRING, amount INT64
) PRIMARY KEY (ckey, skey),
  INTERLEAVE IN PARENT Customer
```

A child's key starts with its parent's, so they sort together. `ckey` is the **sharding key**: shard boundaries are cut on it.



A customer and all its sales sit in one shard.

Example Query

Q1: the running query

```
SELECT ANY_VALUE(c.name) AS name, SUM(s.amount) AS total
FROM   Customer c JOIN Sales s ON c.ckey = s.ckey
WHERE  s.type = 'global' AND c.ckey IN UNNEST(@customer_key_arr)
GROUP  BY c.ckey
ORDER  BY total DESC
LIMIT  5
```

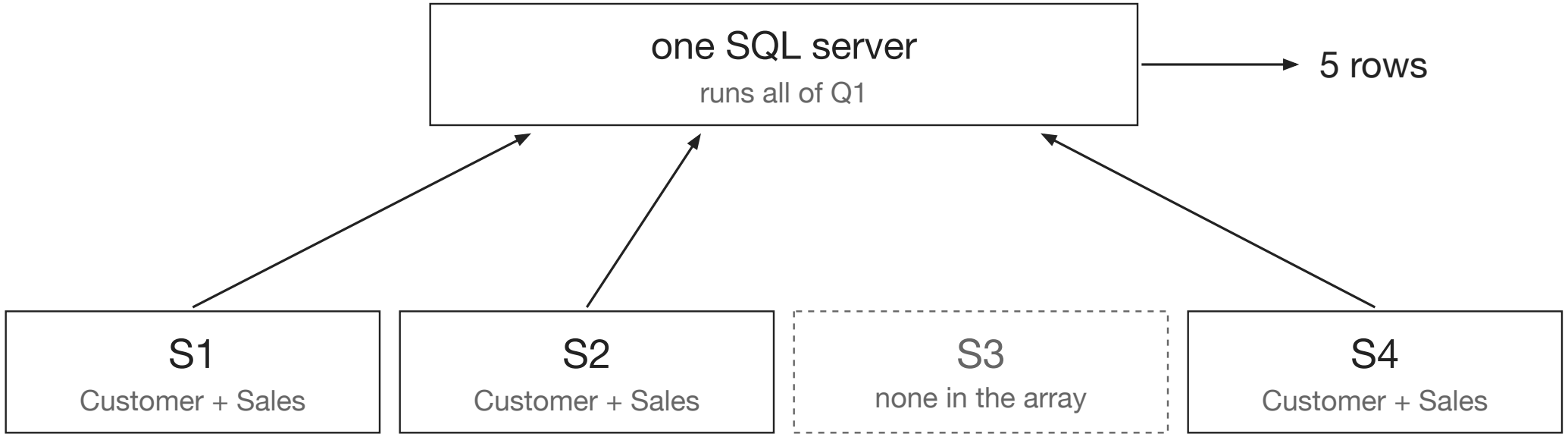
Q1: the running query

```
SELECT ANY_VALUE(c.name) AS name, SUM(s.amount) AS total
FROM   Customer c JOIN Sales s ON c.ckey = s.ckey
WHERE  s.type = 'global' AND c.ckey IN UNNEST(@customer_key_arr)
GROUP  BY c.ckey
ORDER  BY total DESC
LIMIT  5
```

The 5 customers, out of a list passed in (`@arr` for short), with the most `'global'` sales.

Strawman 1: SQL beside the store

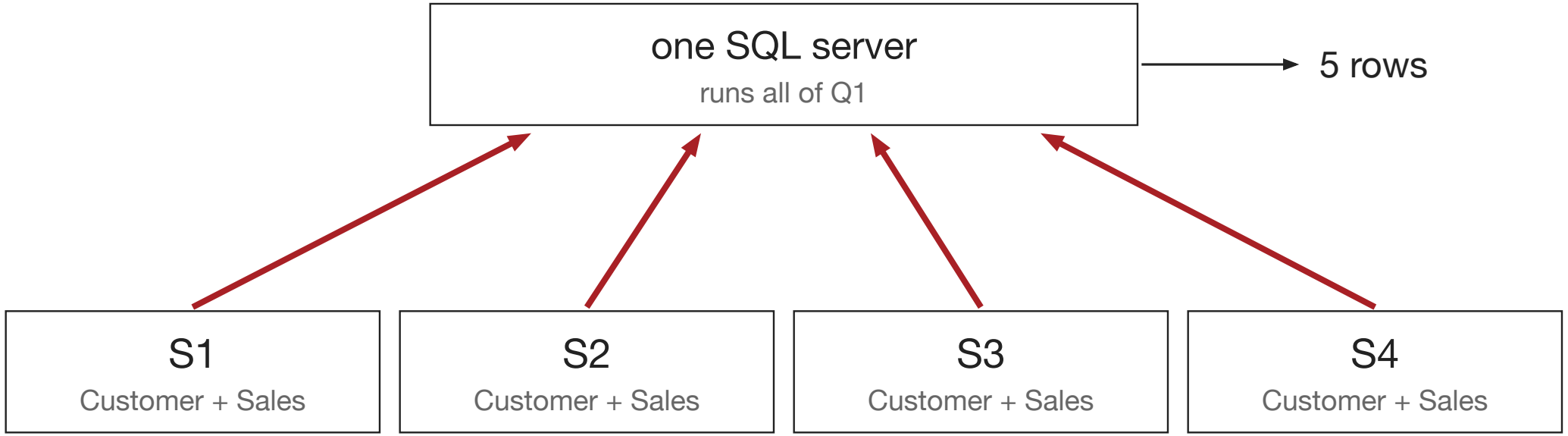
Run all of Q1 on one SQL server; it reads each table through key-range reads against Spanner.



With the array: point reads for its few customers and their sales.

Strawman 1: SQL beside the store

Run all of Q1 on one SQL server; it reads each table through key-range reads against Spanner.



No array, top 5 overall: every Customer and Sales row, global or not.

A key-range read cannot test type.

Ship the operators to the data, not the data to the operators.

Strawman 2: split the plan at compile time

A parallel database partitions once, so its optimizer can write: run this subplan on servers A, B and C; combine their results on one server.

Strawman 2: split the plan at compile time

A parallel database partitions once, so its optimizer can write: run this subplan on servers A, B and C; combine their results on one server.

The compiled plan assumes	By the time it runs
S2 is `[250, 500)`, on G2	S2 split into `[250, 400)` and `[400, 500)`; one half moved

Strawman 2: split the plan at compile time

A parallel database partitions once, so its optimizer can write: run this subplan on servers A, B and C; combine their results on one server.

The compiled plan assumes	By the time it runs
S2 is `[250, 500)`, on G2	S2 split into `[250, 400)` and `[400, 500)`; one half moved
a shard stays put while read	it can move mid-query

Strawman 2: split the plan at compile time

A parallel database partitions once, so its optimizer can write: run this subplan on servers A, B and C; combine their results on one server.

The compiled plan assumes	By the time it runs
S2 is `[250, 500)`, on G2	S2 split into `[250, 400)` and `[400, 500)`; one half moved
a shard stays put while read	it can move mid-query
Q1 reads a known set of shards	which ones depends on `@arr`, unseen at compile time

Plan what runs on each piece; decide which pieces, and where, at run time.

Distribution as an operator

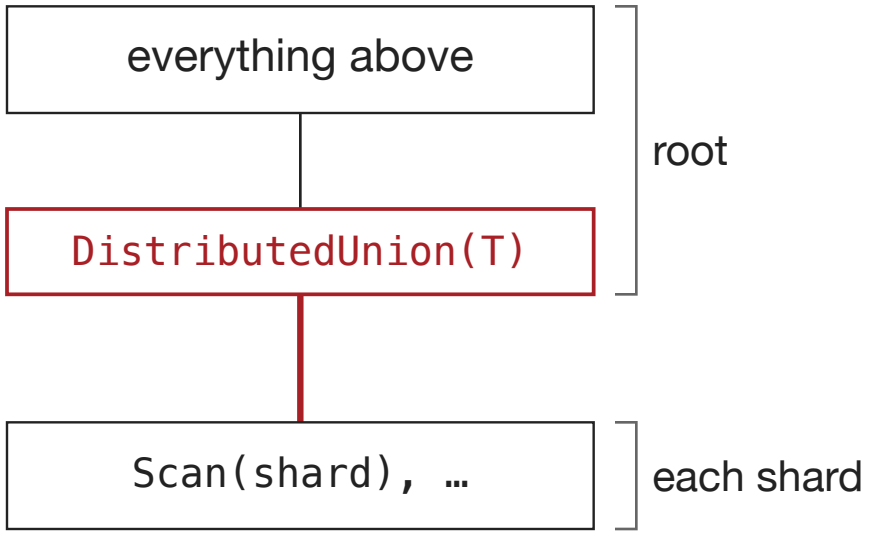
DistributedUnion

```
Scan(T) ⇒ DistributedUnion[shard ⊆ T](Scan(shard))
```

DistributedUnion

```
Scan(T) ⇒ DistributedUnion[shard ⊆ T](Scan(shard))
```

compile time: one tree

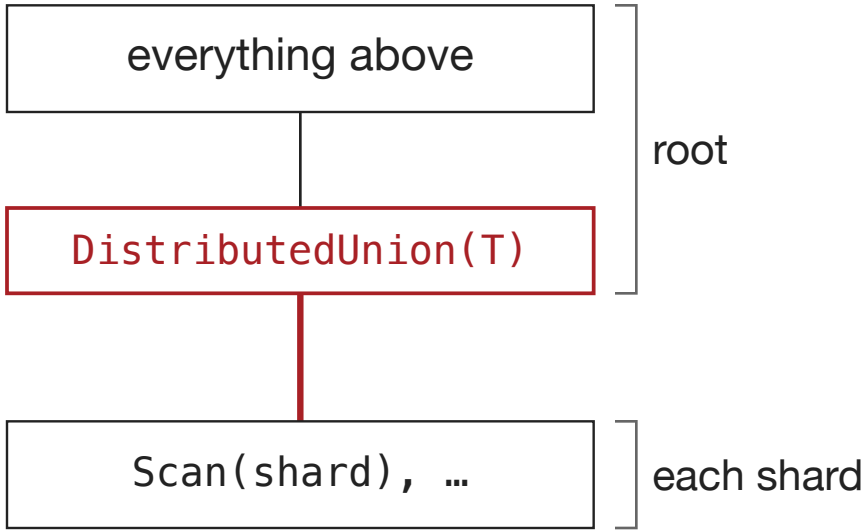


The **root server**, the one the query was sent to, runs everything above the outermost union.

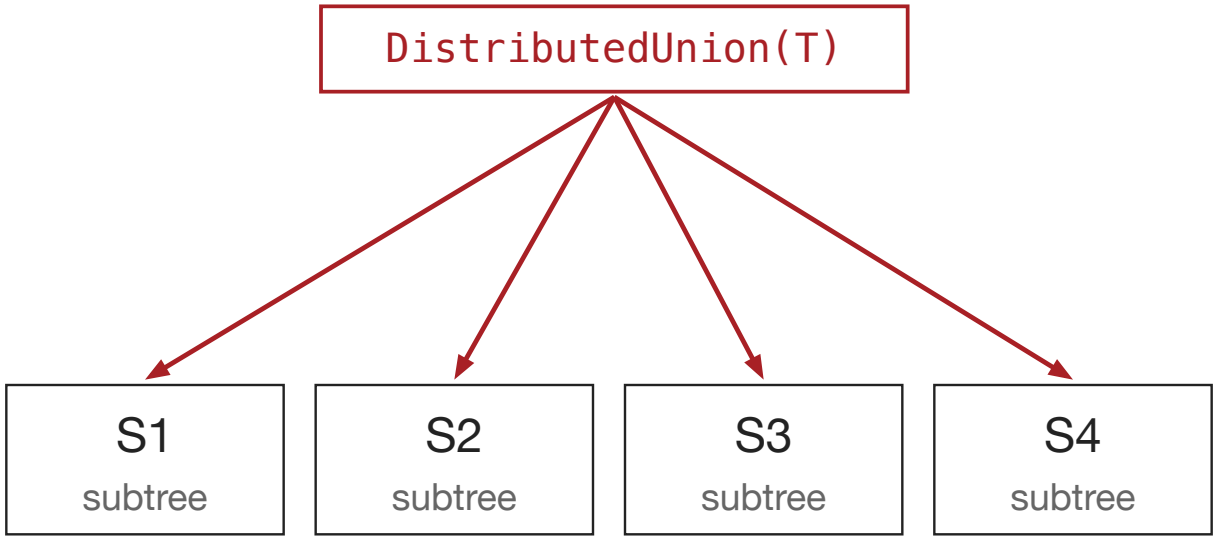
DistributedUnion

```
Scan(T) ⇒ DistributedUnion[shard ⊆ T](Scan(shard))
```

compile time: one tree



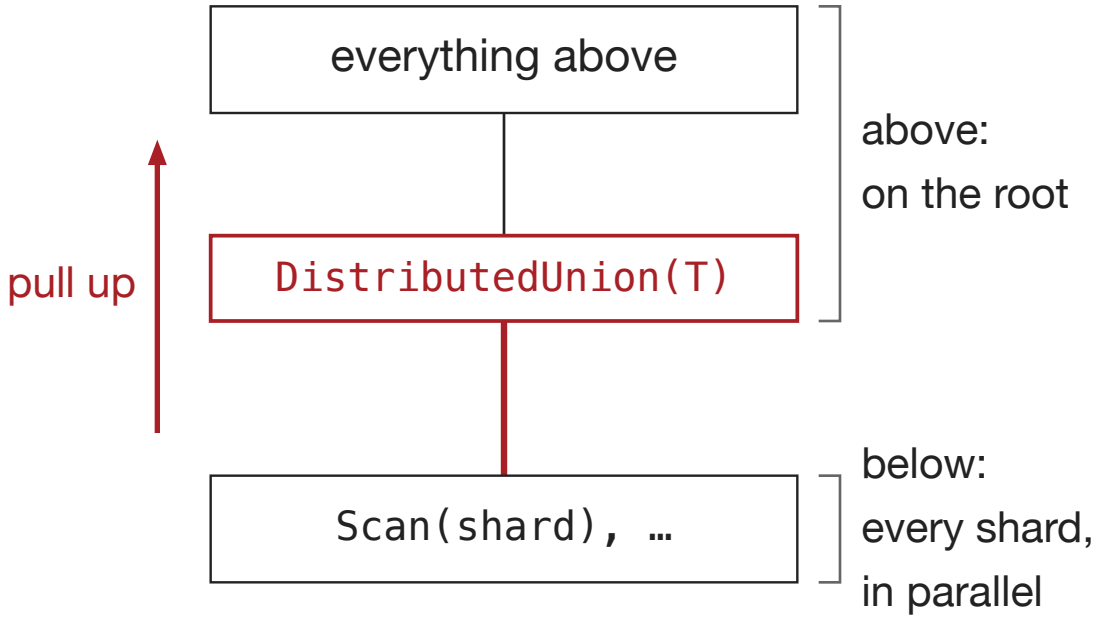
run time: 1-to-N calls



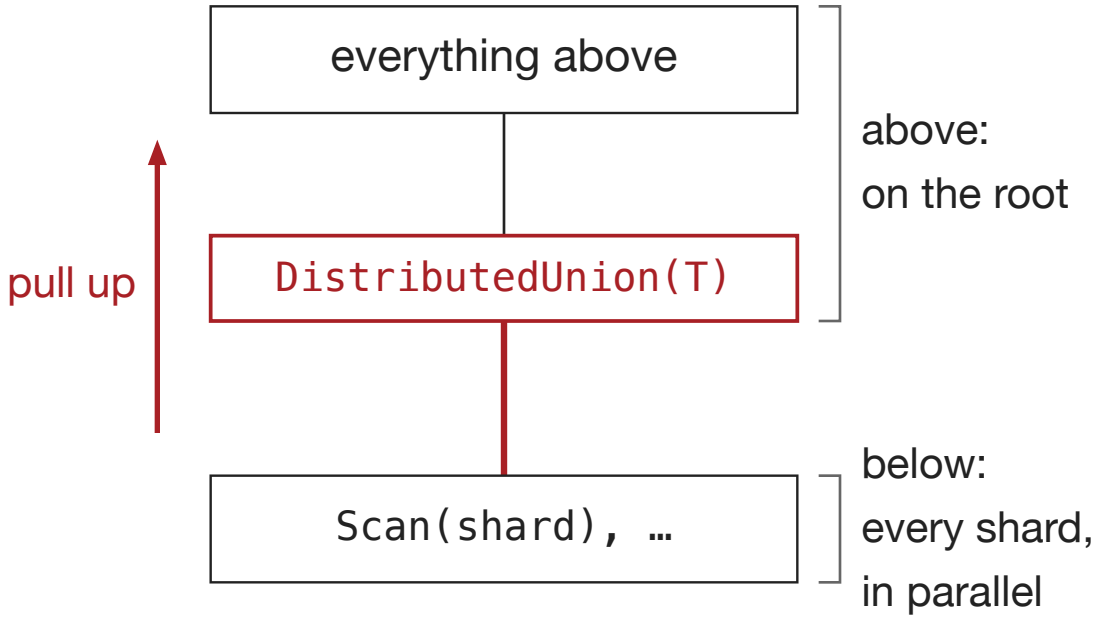
results come back and are concatenated

The **root server**, the one the query was sent to, runs everything above the outermost union.

Pull the union up

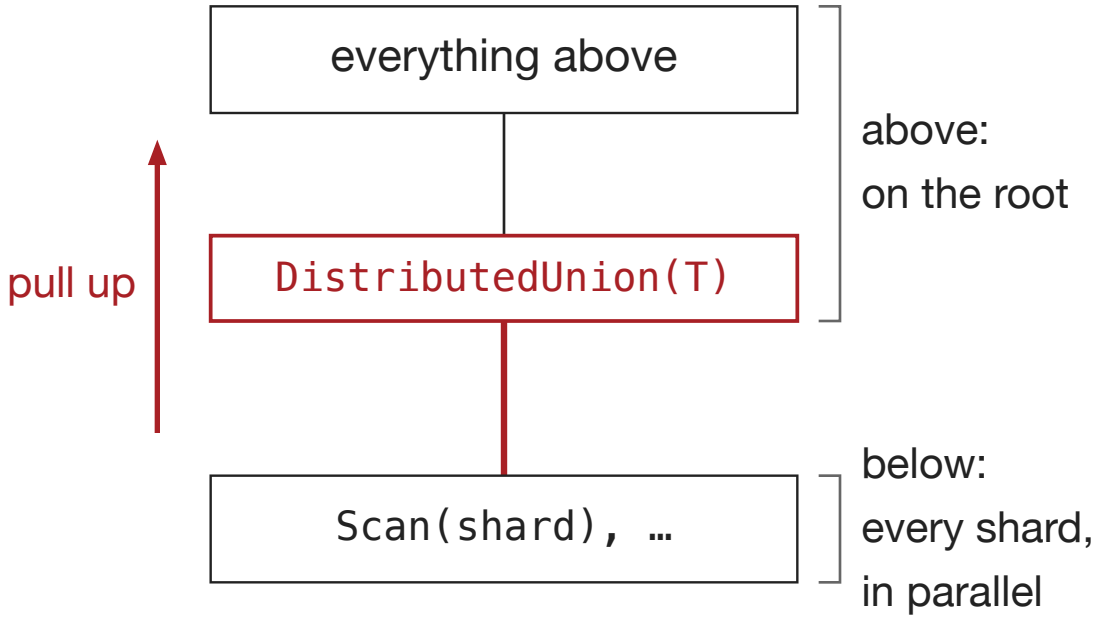


Pull the union up



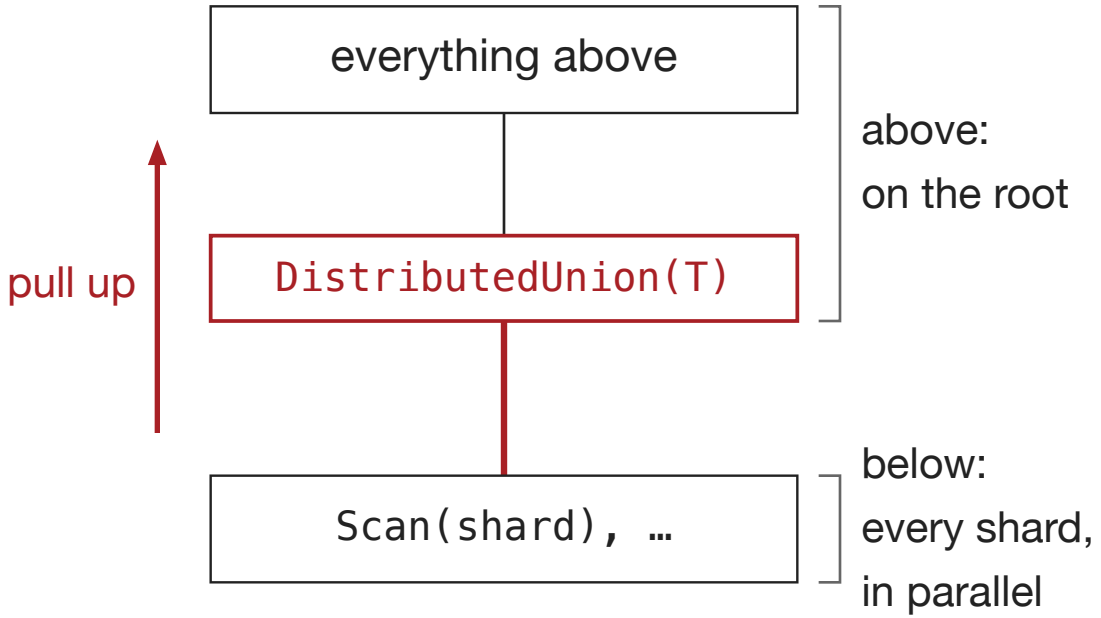
- Last lecture promised cost terms for network traffic and for shards called.

Pull the union up



- Last lecture promised cost terms for network traffic and for shards called.
- Here, rules instead: pull the union up wherever it may go, and cut calls.

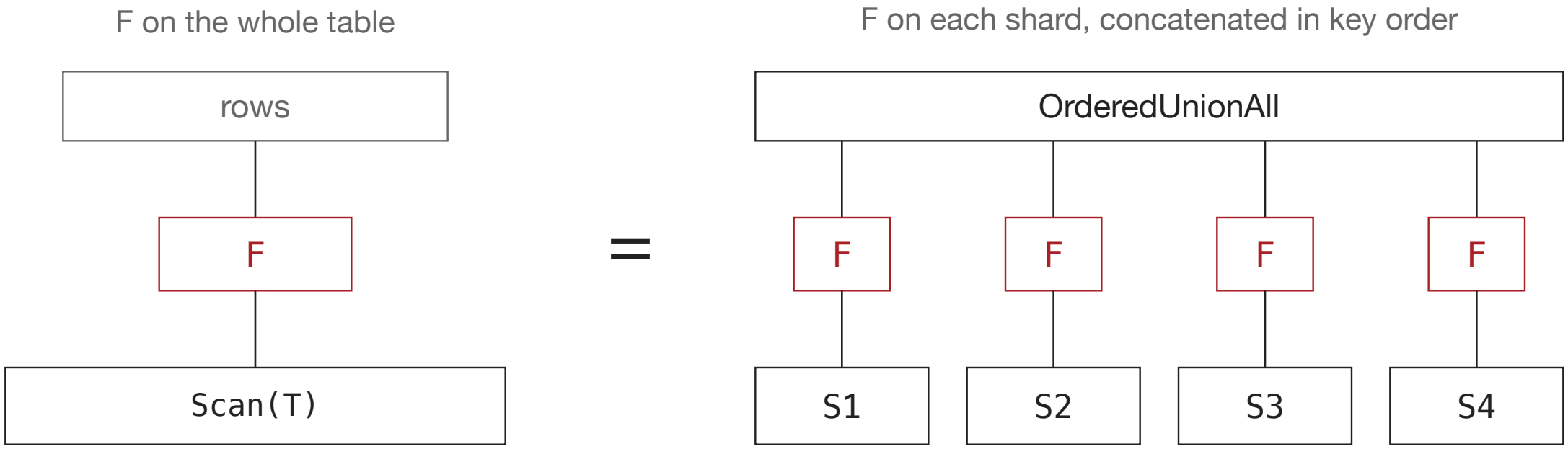
Pull the union up



- Last lecture promised cost terms for network traffic and for shards called.
- Here, rules instead: pull the union up wherever it may go, and cut calls.
- Join order, statistics: out of scope today.

Partitionability

```
F(Scan(T)) = OrderedUnionAll[shard ⊆ T](F(Scan(shard)))
```



Pulling the union above F is correct only if this holds.

OrderedUnionAll: concatenate the shards' results in key order.

Which operators pass?

Customer and Sales, sharded on `ckey`. Per shard, then concatenate: the same answer?

Operator	Passes?	Why
Filter, Project	?	
`GroupBy(ckey)`	?	
`GroupBy(type)`	?	
Sort by `ckey`, ascending	?	
`Top(5)`, by any sort key	?	

`Top(5)` is `ORDER BY … LIMIT 5` as one operator.

Which operators pass?

Customer and Sales, sharded on `ckey`. Per shard, then concatenate: the same answer?

Operator	Passes?	Why
Filter, Project	passes	one row at a time
`GroupBy(ckey)`	?	
`GroupBy(type)`	?	
Sort by `ckey`, ascending	?	
`Top(5)`, by any sort key	?	

`Top(5)` is `ORDER BY … LIMIT 5` as one operator.

Which operators pass?

Customer and Sales, sharded on `ckey`. Per shard, then concatenate: the same answer?

Operator	Passes?	Why
Filter, Project	passes	one row at a time
`GroupBy(ckey)`	passes	a ckey's rows never span a boundary
`GroupBy(type)`	?	
Sort by `ckey`, ascending	?	
`Top(5)`, by any sort key	?	

`Top(5)` is `ORDER BY … LIMIT 5` as one operator.

Which operators pass?

Customer and Sales, sharded on `ckey`. Per shard, then concatenate: the same answer?

Operator	Passes?	Why
Filter, Project	passes	one row at a time
`GroupBy(ckey)`	passes	a ckey's rows never span a boundary
`GroupBy(type)`	fails	one type's rows span the shards
Sort by `ckey`, ascending	?	
`Top(5)`, by any sort key	?	

`Top(5)` is `ORDER BY … LIMIT 5` as one operator.

Which operators pass?

Customer and Sales, sharded on `ckey`. Per shard, then concatenate: the same answer?

Operator	Passes?	Why
Filter, Project	passes	one row at a time
`GroupBy(ckey)`	passes	a ckey's rows never span a boundary
`GroupBy(type)`	fails	one type's rows span the shards
Sort by `ckey`, ascending	passes	sort each shard, concatenate in key order
`Top(5)`, by any sort key	?	

`Top(5)` is `ORDER BY … LIMIT 5` as one operator.

Which operators pass?

Customer and Sales, sharded on `ckey`. Per shard, then concatenate: the same answer?

Operator	Passes?	Why
Filter, Project	passes	one row at a time
`GroupBy(ckey)`	passes	a ckey's rows never span a boundary
`GroupBy(type)`	fails	one type's rows span the shards
Sort by `ckey`, ascending	passes	sort each shard, concatenate in key order
`Top(5)`, by any sort key	fails	N shards keep 5 each: up to 5N rows

Groups and orders that follow the sharding key pass; Top never does.

`Top(5)` is `ORDER BY … LIMIT 5` as one operator.

Split what does not pass

```
Op( DistributedUnion[shard ⊆ T]( F(Scan(shard)) ) )
  = OpFinal( DistributedUnion[shard ⊆ T]( OpLocal( F(Scan(shard)) ) ) )
```

Op	OpLocal, on each shard	OpFinal, above the union
`Top(5)`		
`SUM`		
`AVG`		

Split what does not pass

```
Op( DistributedUnion[shard ⊆ T]( F(Scan(shard)) ) )
  = OpFinal( DistributedUnion[shard ⊆ T]( OpLocal( F(Scan(shard)) ) ) )
```

Op	OpLocal, on each shard	OpFinal, above the union
`Top(5)`	its own top 5	top 5 of the shards' top 5s
`SUM`		
`AVG`		

Top is right: each of the table's top 5 rows is in its own shard's top 5.

Split what does not pass

```
Op( DistributedUnion[shard ⊆ T]( F(Scan(shard)) ) )
  = OpFinal( DistributedUnion[shard ⊆ T]( OpLocal( F(Scan(shard)) ) ) )
```

Op	OpLocal, on each shard	OpFinal, above the union
`Top(5)`	its own top 5	top 5 of the shards' top 5s
`SUM`	`SUM`	`SUM` of the sums
`AVG`		

Top is right: each of the table's top 5 rows is in its own shard's top 5.

Split what does not pass

```
Op( DistributedUnion[shard ⊆ T]( F(Scan(shard)) ) )
  = OpFinal( DistributedUnion[shard ⊆ T]( OpLocal( F(Scan(shard)) ) ) )
```

Op	OpLocal, on each shard	OpFinal, above the union
`Top(5)`	its own top 5	top 5 of the shards' top 5s
`SUM`	`SUM`	`SUM` of the sums
`AVG`	`SUM` and `COUNT`	`SUM` of sums ÷ `SUM` of counts

Ship a small summary per shard; finish above the union.

Top is right: each of the table's top 5 rows is in its own shard's top 5.

Joins as Apply

`CrossApply(input, map)`: for each row of input, run map with that row's columns bound; concatenate the results.

```
CrossApply(input, map):
  for each row c of input:
    for each row s of map(c):    -- c's columns are constants here
      emit (c, s)
```

In the map, `s.ckey = c.ckey` becomes `s.ckey = 7`: a seek to key 7 in Sales, not a scan.

`OuterApply` is the left-outer version: NULLs when the map returns nothing.

Parent-child joins: two unions merge

```
S1  Customer(7)
      Sales(7, 1)
      Sales(7, 2)
```

```
S2  Customer(300)
      Sales(300, 1)
      Sales(300, 2)
```

Sales is interleaved in Customer:
each sale sits in its customer's shard,
so no match is ever in another shard.

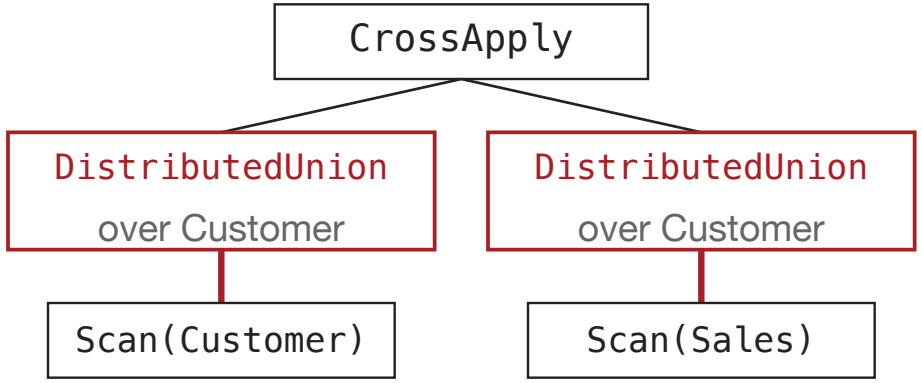
Parent-child joins: two unions merge

```
S1  Customer(7)
      Sales(7, 1)
      Sales(7, 2)
```

```
S2  Customer(300)
      Sales(300, 1)
      Sales(300, 2)
```

Sales is interleaved in Customer: each sale sits in its customer's shard, so no match is ever in another shard.

before: a union under each input

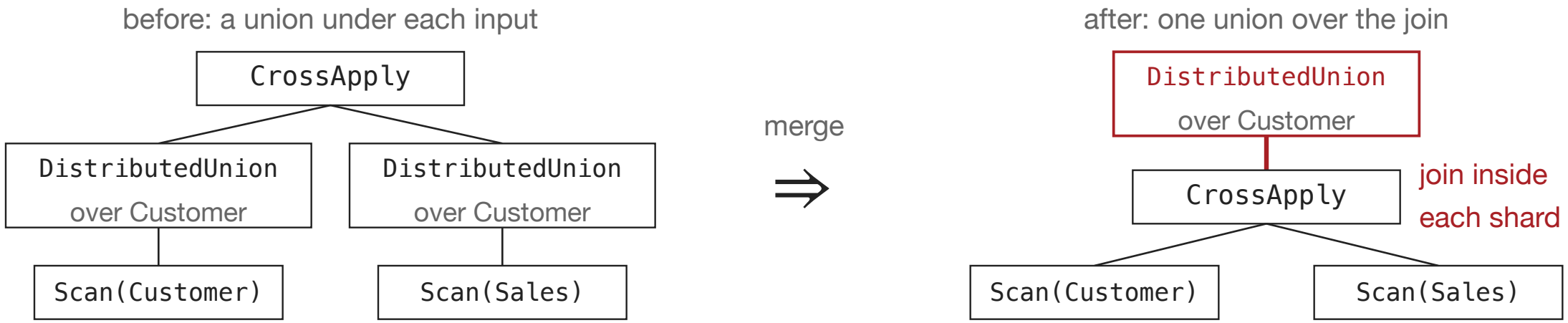


Parent-child joins: two unions merge

```
S1  Customer(7)
      Sales(7, 1)
      Sales(7, 2)
```

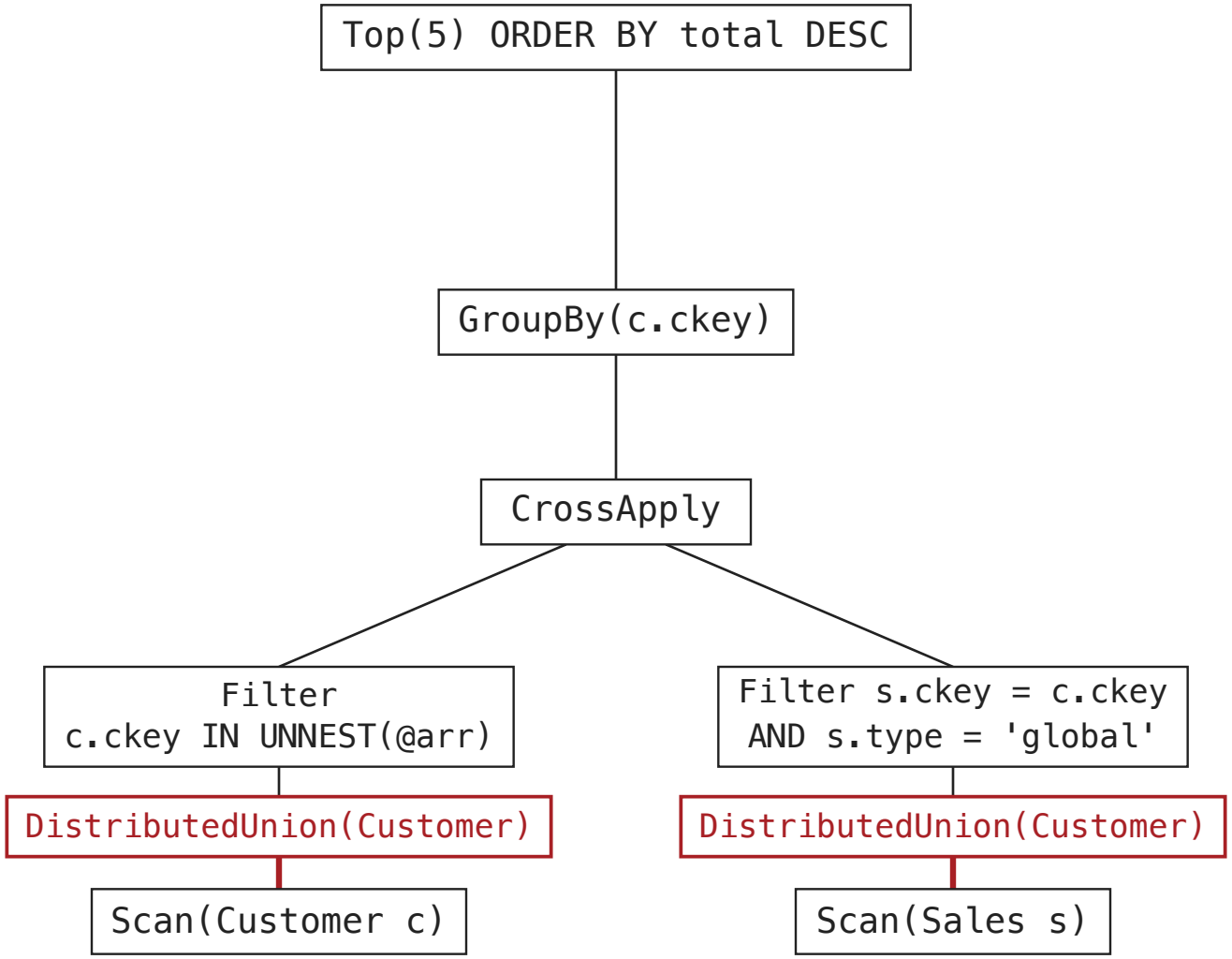
```
S2  Customer(300)
      Sales(300, 1)
      Sales(300, 2)
```

Sales is interleaved in Customer:
each sale sits in its customer's shard,
so no match is ever in another shard.



The parent-child join runs inside each shard.

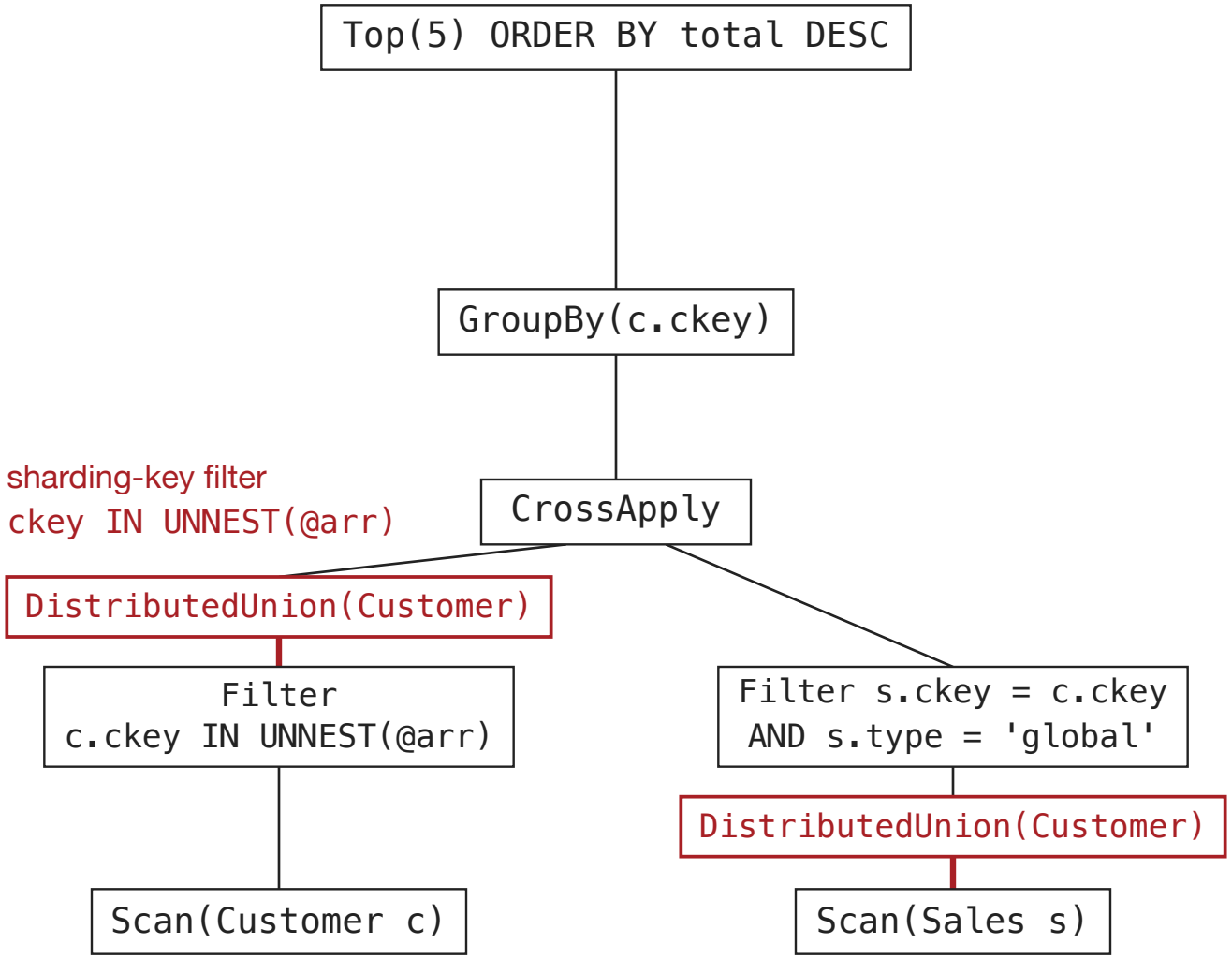
Q1: pull the union up



Start: a union over each scan.

Sales is interleaved: Customer's union.

Q1: pull the union up



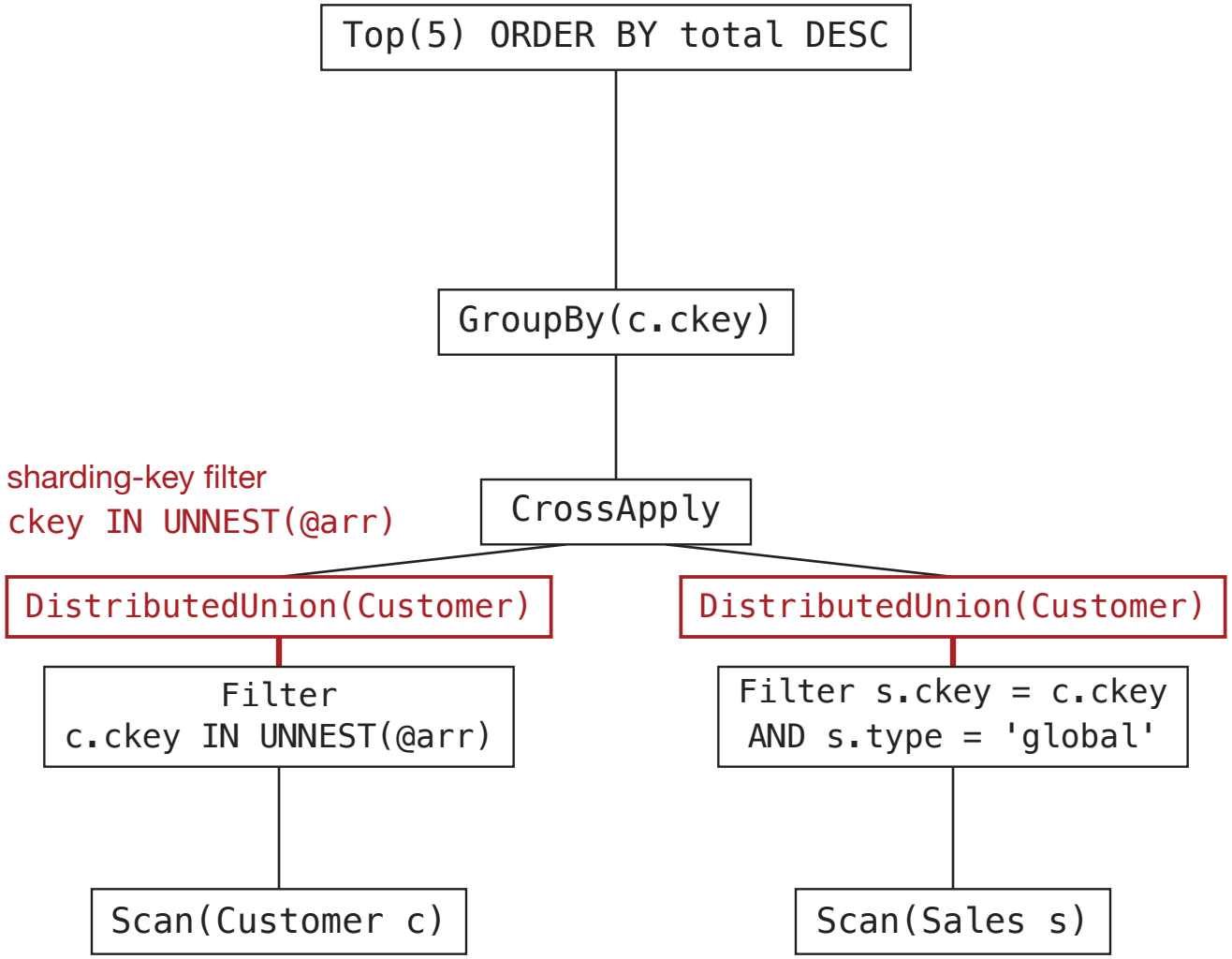
Start: a union over each scan.

Sales is interleaved: Customer's union.

1 · Filter on c: passes.

Its ckey test: the sharding-key filter.

Q1: pull the union up



Start: a union over each scan.

Sales is interleaved: Customer's union.

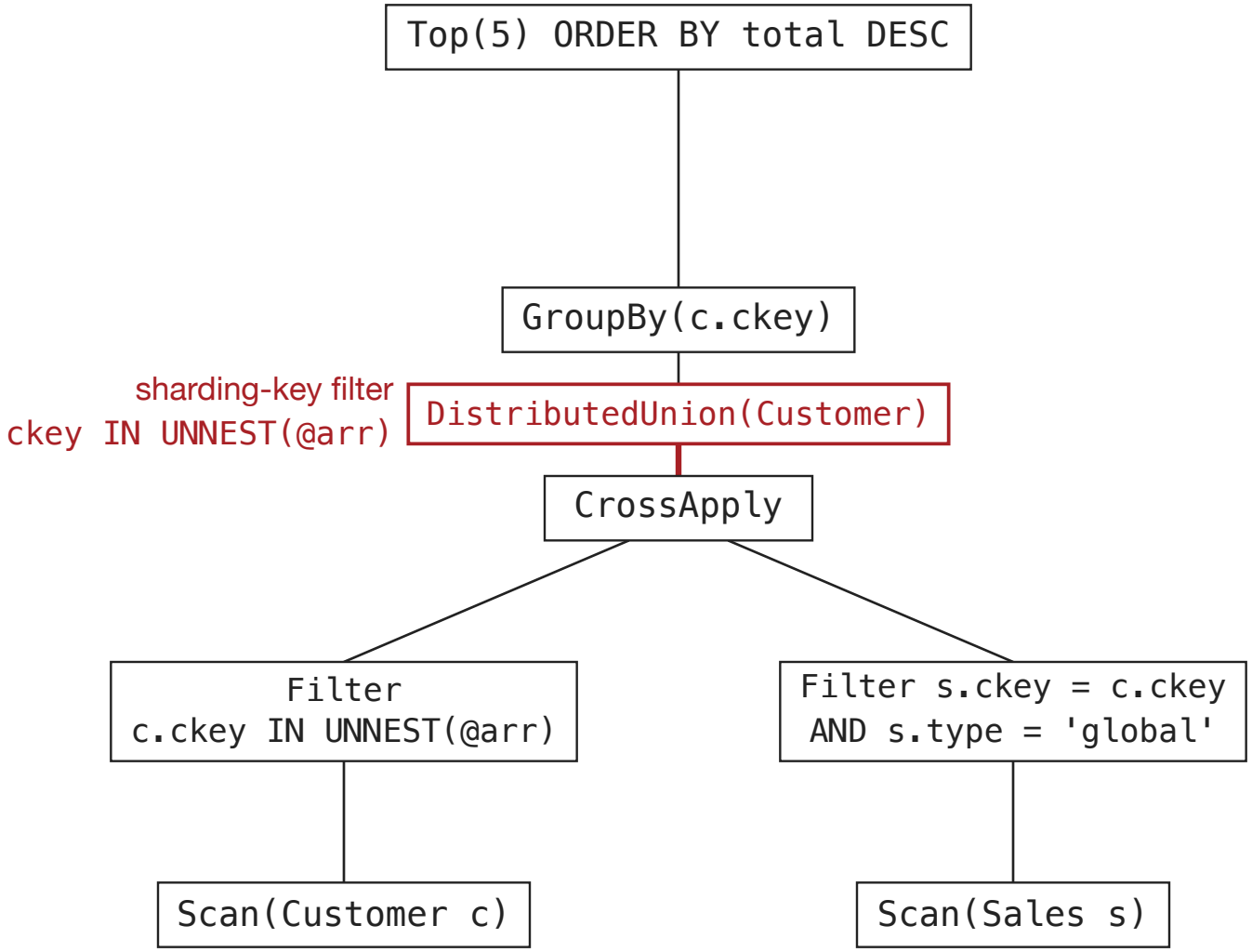
1 · Filter on c: passes.

Its ckey test: the sharding-key filter.

2 · Filter on s: passes.

Its sharding-key filter: s.ckey = c.ckey.

Q1: pull the union up



Start: a union over each scan.

Sales is interleaved: Customer's union.

1 · Filter on c: passes.

Its ckey test: the sharding-key filter.

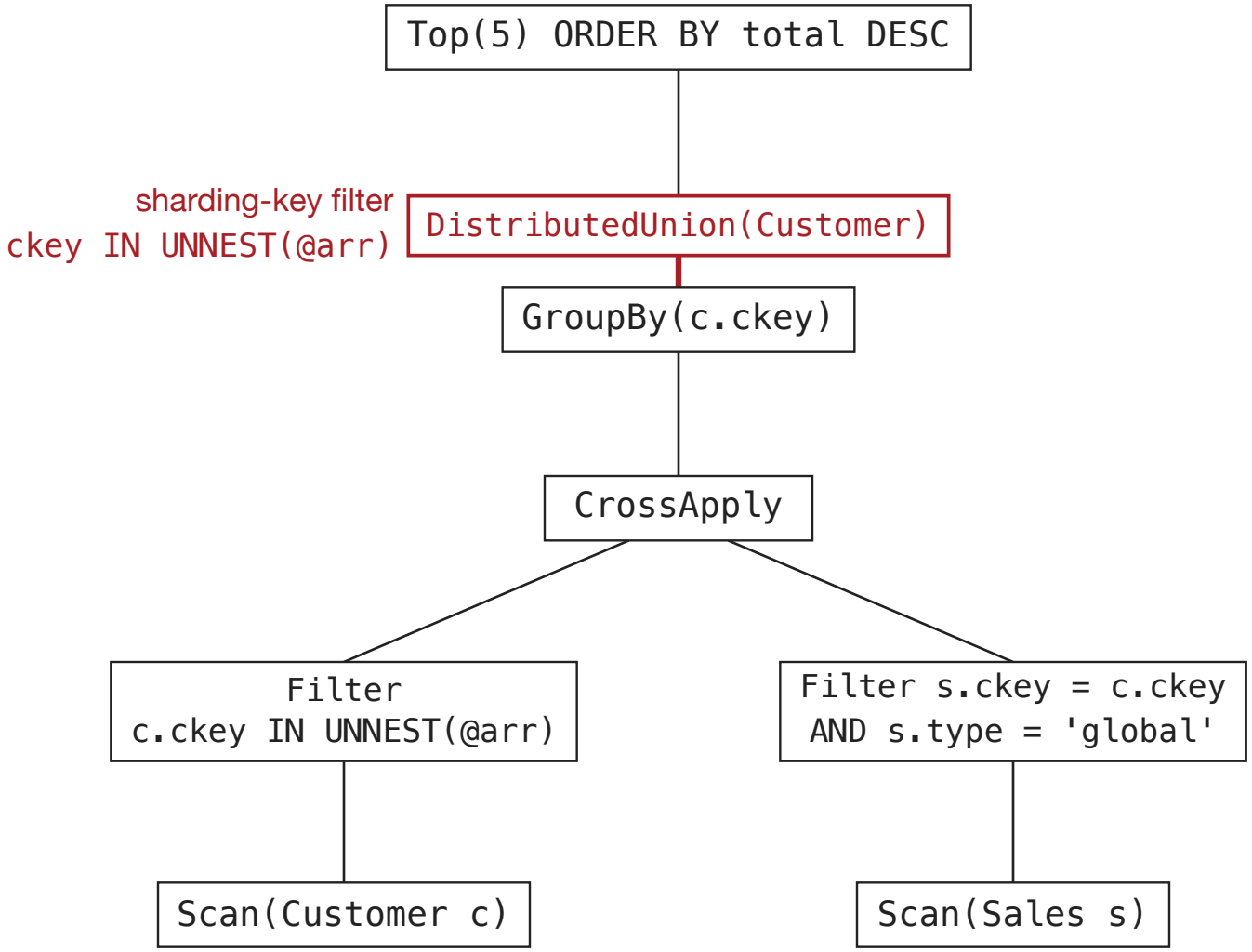
2 · Filter on s: passes.

Its sharding-key filter: s.ckey = c.ckey.

3 · CrossApply: merge.

Same table, joined on all of ckey.

Q1: pull the union up



Start: a union over each scan.

Sales is interleaved: Customer's union.

1 · Filter on c: passes.

Its ckey test: the sharding-key filter.

2 · Filter on s: passes.

Its sharding-key filter: s.ckey = c.ckey.

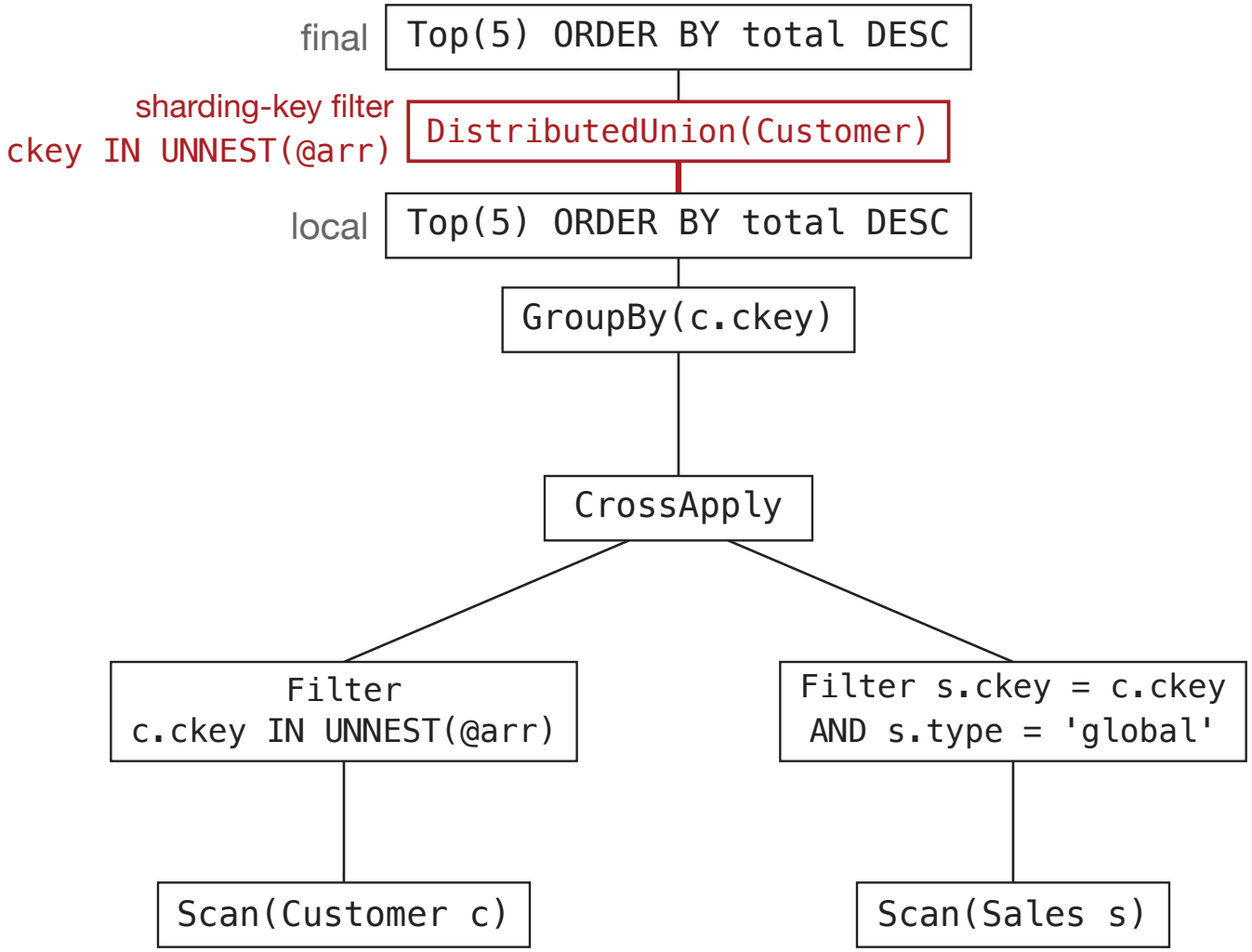
3 · CrossApply: merge.

Same table, joined on all of ckey.

4 · GroupBy(c.ckey): passes.

A customer's rows share one shard.

Q1: pull the union up



Start: a union over each scan.

Sales is interleaved: Customer's union.

1 · Filter on c: passes.

Its ckey test: the sharding-key filter.

2 · Filter on s: passes.

Its sharding-key filter: s.ckey = c.ckey.

3 · CrossApply: merge.

Same table, joined on all of ckey.

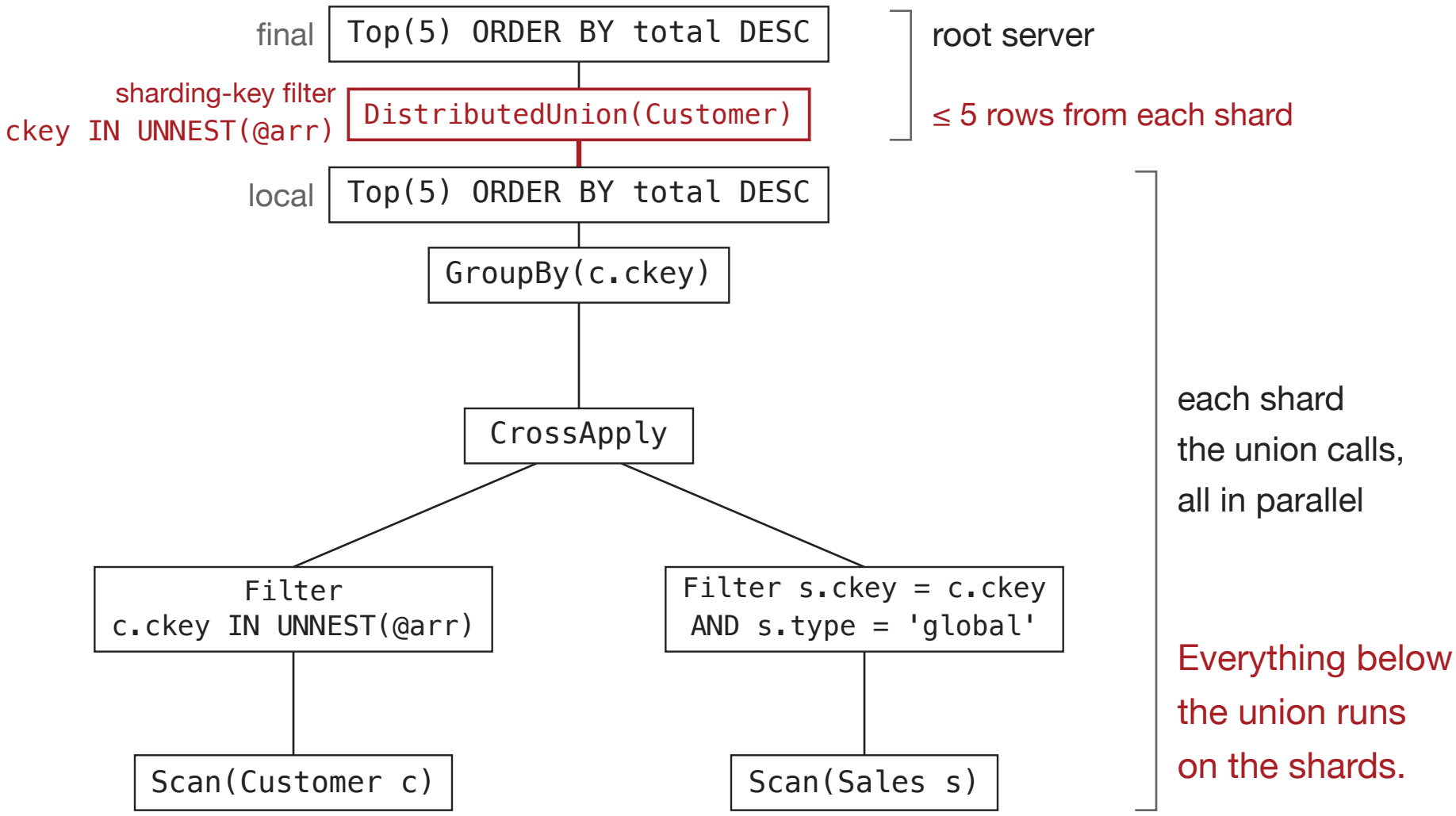
4 · GroupBy(c.ckey): passes.

A customer's rows share one shard.

5 · Top(5) by total: fails.

Local Top below, final Top above.

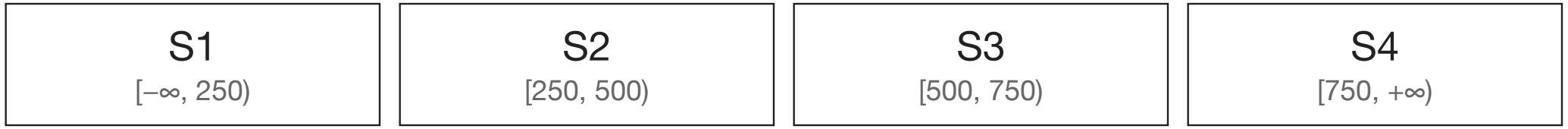
Q1, distributed



Which shards: pruning

The union evaluates its filter with this run's `@arr`, before calling anyone.

```
@arr = [7, 412, 7, 950]
```

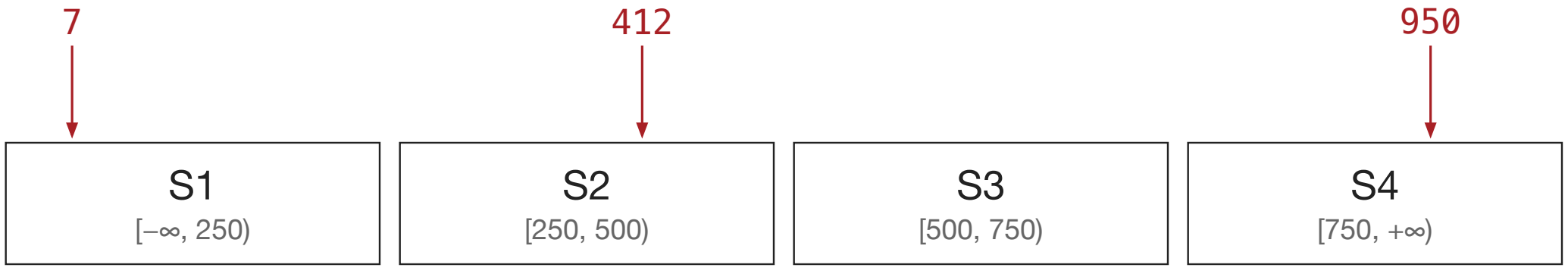


Which shards: pruning

The union evaluates its filter with this run's `@arr`, before calling anyone.

```
@arr = [7, 412, 7, 950]
```

remove duplicates: the points 7, 412 and 950

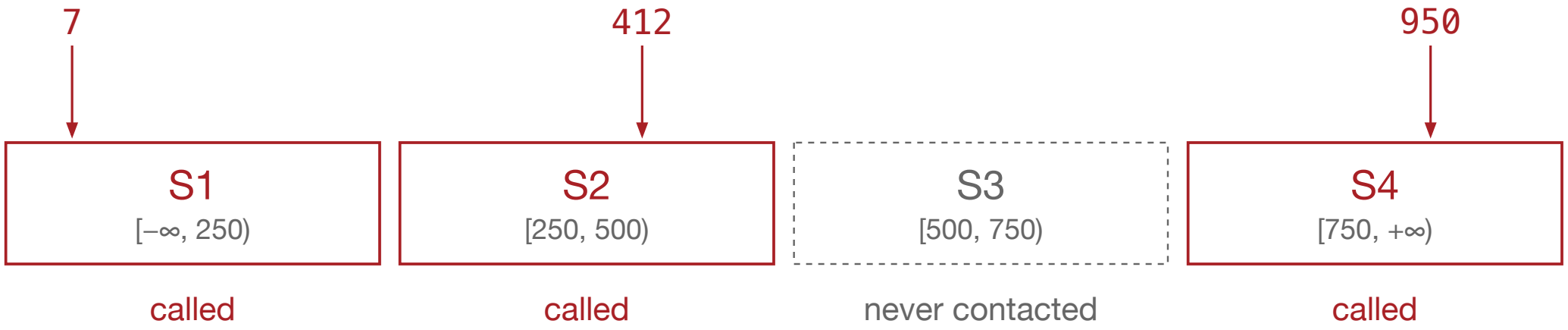


Which shards: pruning

The union evaluates its filter with this run's `@arr`, before calling anyone.

```
@arr = [7, 412, 7, 950]
```

remove duplicates: the points 7, 412 and 950



S3 is never contacted: **shard pruning.**

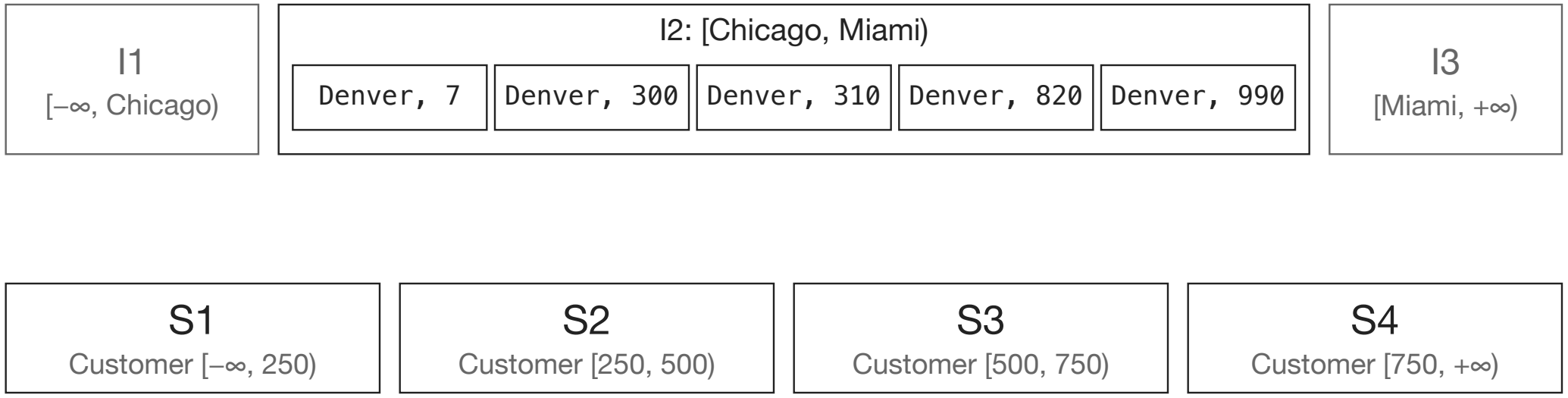
Safe: every row the subtree could return passes the filter, so it lies in a range we call.

Joins across shards

A secondary index is its own table

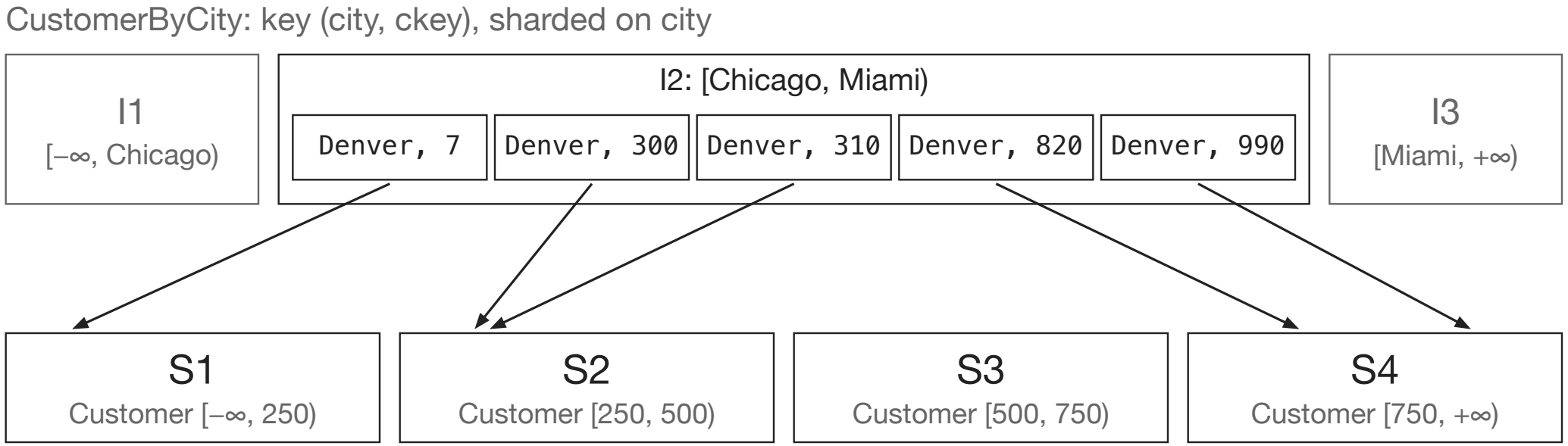
Q2: `SELECT c.* FROM Customer c WHERE c.city = 'Denver'`

CustomerByCity: key (city, ckey), sharded on city



A secondary index is its own table

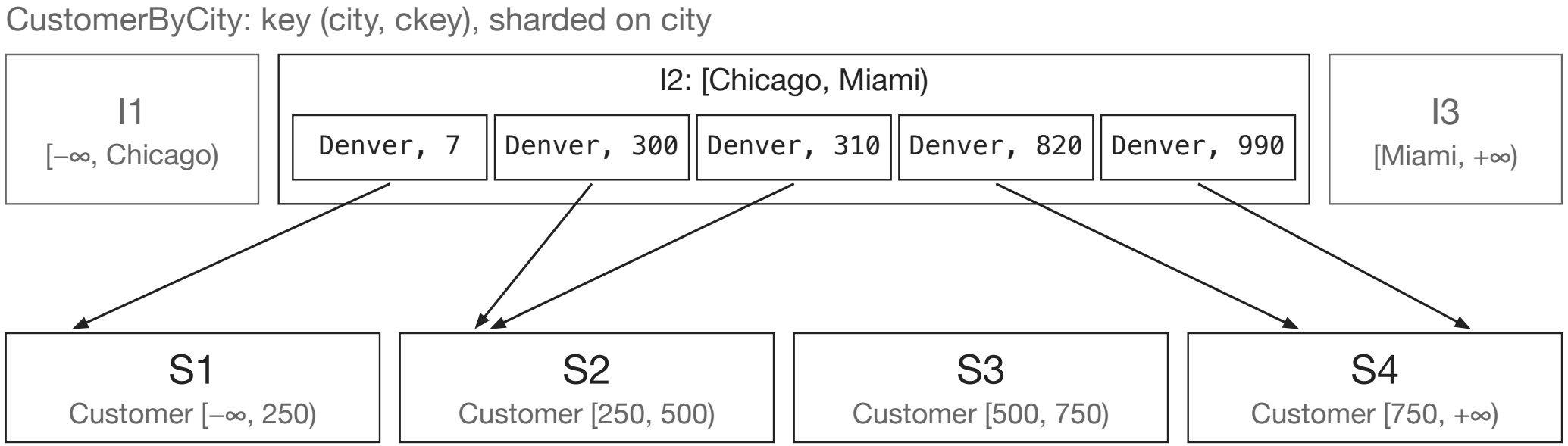
Q2: `SELECT c.* FROM Customer c WHERE c.city = 'Denver'`



One index shard holds Denver; its rows are in three Customer shards.

A secondary index is its own table

Q2: `SELECT c.* FROM Customer c WHERE c.city = 'Denver'`

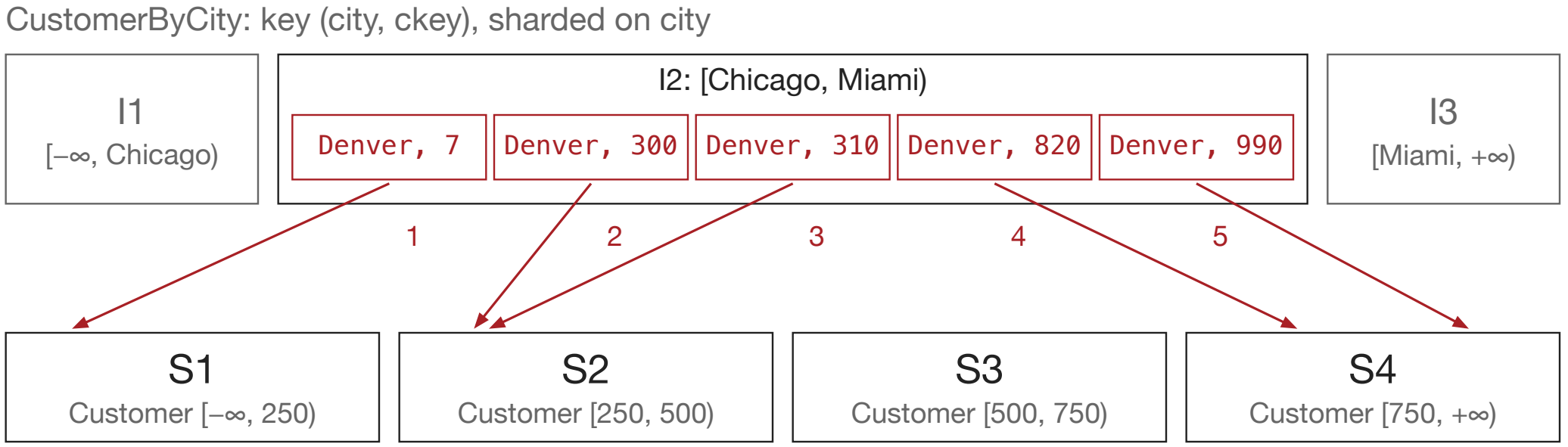


One index shard holds Denver; its rows are in three Customer shards.

```
SELECT c.* FROM CustomerByCity i JOIN Customer c ON c.ckey = i.ckey
WHERE  i.city = 'Denver'      -- Q2 with the index as a table: a join
```

One call per row

As written: a `CrossApply`, input the index scan, map the Customer lookup, a union over each.

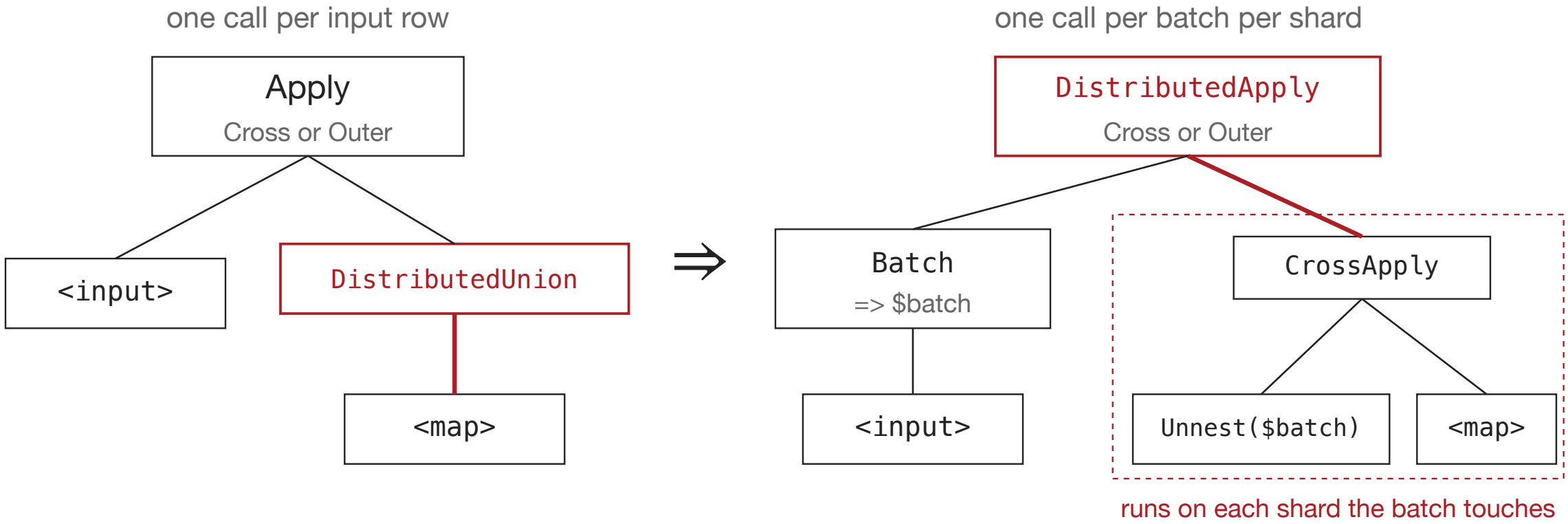


5 rows, 5 calls, one after another

Nested-loop cost: `C = C_outer + outer_rows × C_inner`, outer the index scan, inner one map lookup. The map's union cannot be pulled up: the tables are sharded differently. So `C_inner` is a call.

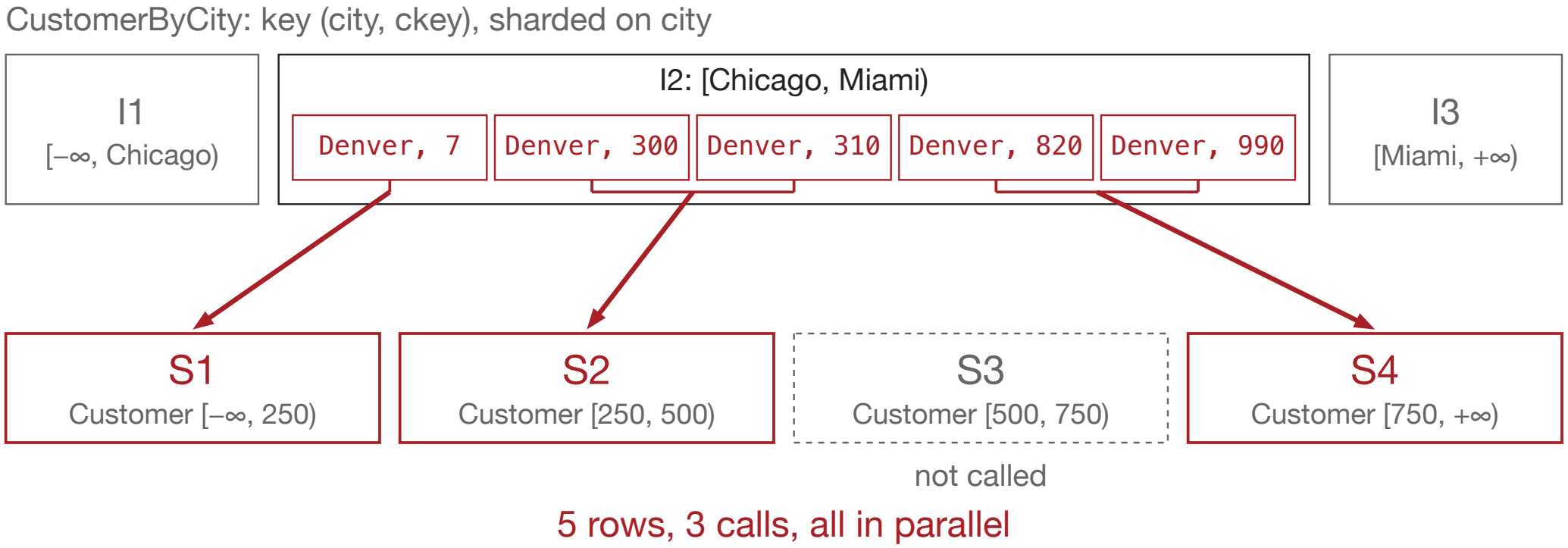
Batch the input: DistributedApply

One join becomes two: a distributed one ships batches; a local one applies each row.



Routing one batch

Each row's ckey picks its shard; each shard gets only its rows, in one call.



The index now pays per shard touched, not per row.

Range extraction

Which keys can a predicate touch?

```
Documents  PRIMARY KEY (ProjectId, DocumentPath, Version)
           sharded on ProjectId
SELECT d.* FROM Documents d
WHERE  d.ProjectId = @param1
  AND  STARTS_WITH(d.DocumentPath, '/proposals')
  AND  d.Version = @param2
```

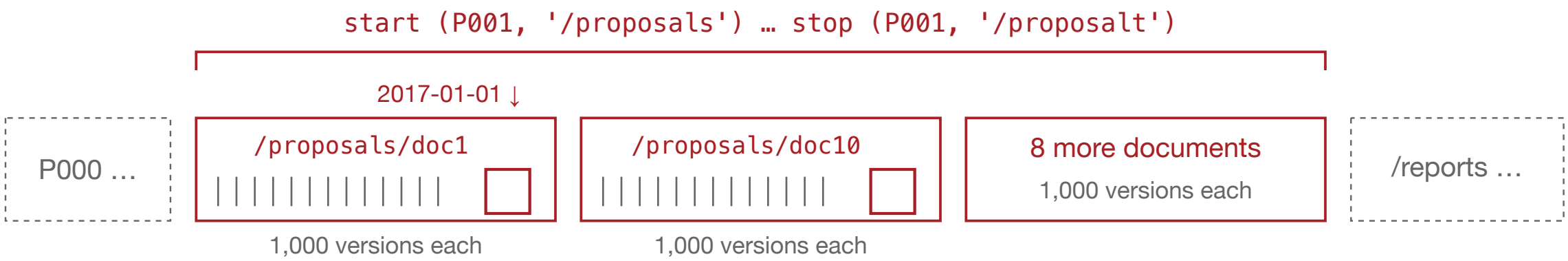
Which shards to call, what to read, what to lock.

`@param1` = P001, `@param2` = 2017-01-01; ten documents under /proposals, 1,000 versions each.

Strawman: one start key, one stop key

All three conditions test key columns, but only equalities, then one range, narrow a scan.

Documents in key order: (ProjectId, DocumentPath, Version)



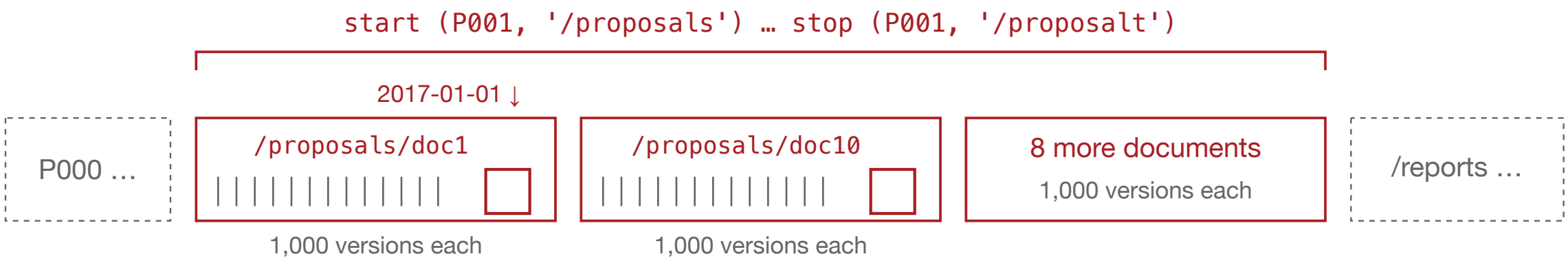
one range: 10 × 1,000 = 10,000 rows read, 10 kept

Version = `@param2` is tested on every row; it narrows nothing.

Strawman: one start key, one stop key

All three conditions test key columns, but only equalities, then one range, narrow a scan.

Documents in key order: (ProjectId, DocumentPath, Version)



one range: 10 × 1,000 = 10,000 rows read, 10 kept

Version = `@param2` is tested on every row; it narrows nothing.

Wanted: one seek per document, to its 2017-01-01 version.

Key columns one at a time

Pin one key column, then constrain the next given its value: one scan per key column.

```
Scan1(Documents.ProjectId)
filter ProjectId = @param1
emits @project_id: no read
```

Scan1 handles the first key column, ProjectId: the filter pins it to P001.

Key columns one at a time

Pin one key column, then constrain the next given its value: one scan per key column.

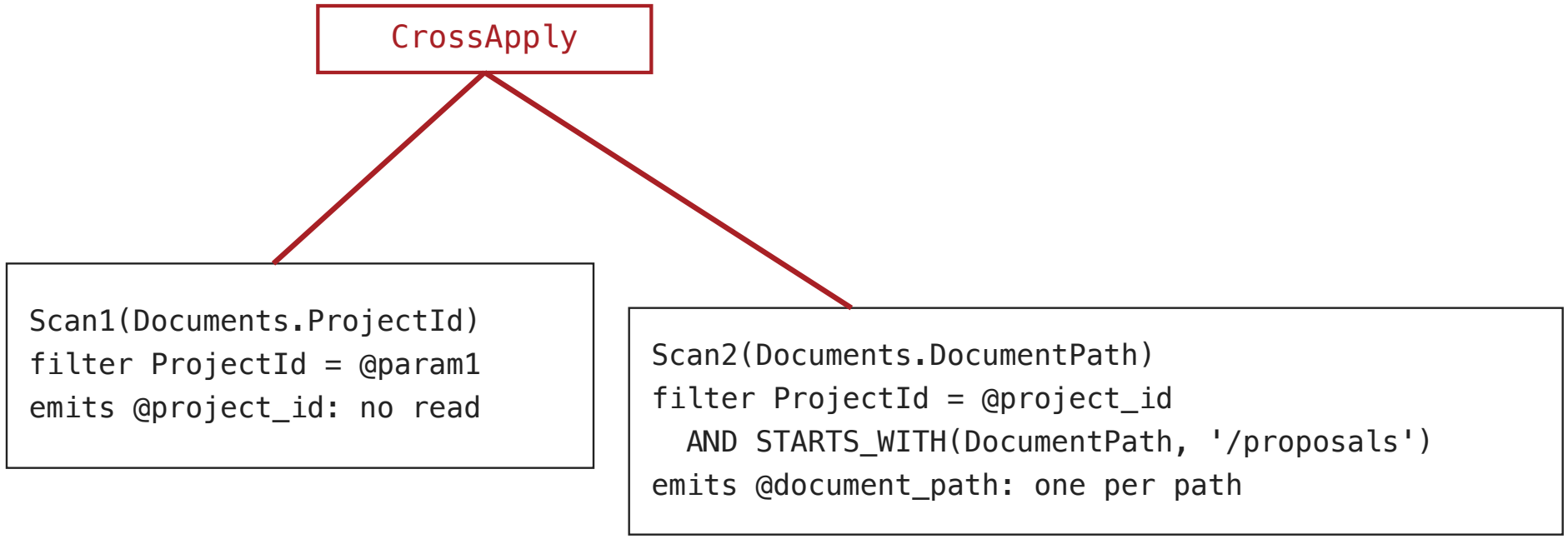
```
Scan1(Documents.ProjectId)
filter ProjectId = @param1
emits @project_id: no read
```

```
Scan2(Documents.DocumentPath)
filter ProjectId = @project_id
  AND STARTS_WITH(DocumentPath, '/proposals')
emits @document_path: one per path
```

Scan2 handles DocumentPath, but needs a `@project_id` from somewhere.

Key columns one at a time

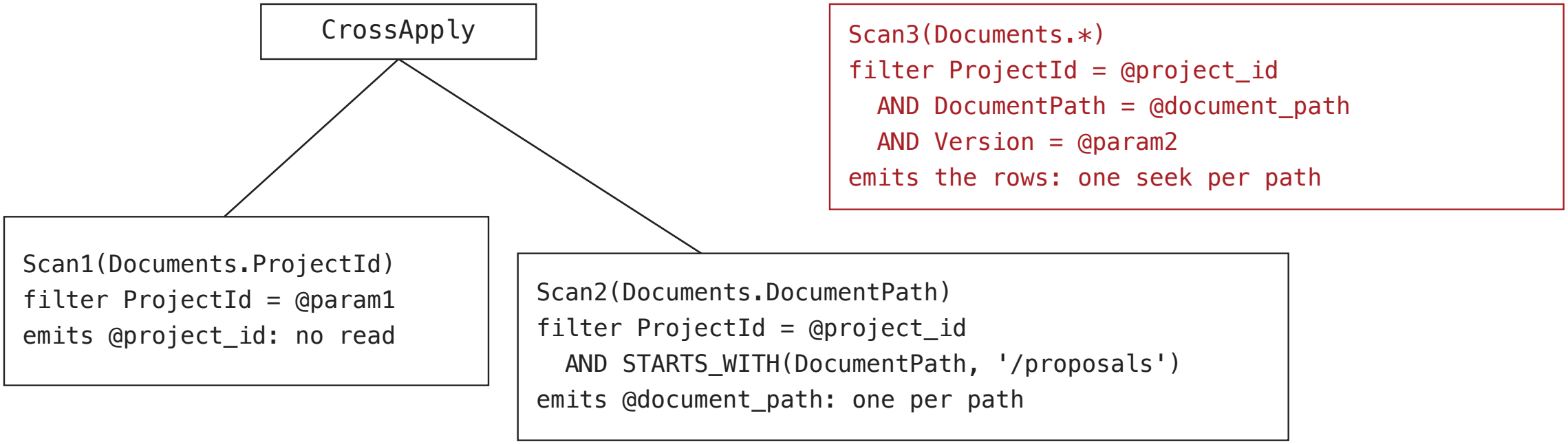
Pin one key column, then constrain the next given its value: one scan per key column.



For each `@project_id` Scan1 emits (only P001), run Scan2: it emits the ten paths.

Key columns one at a time

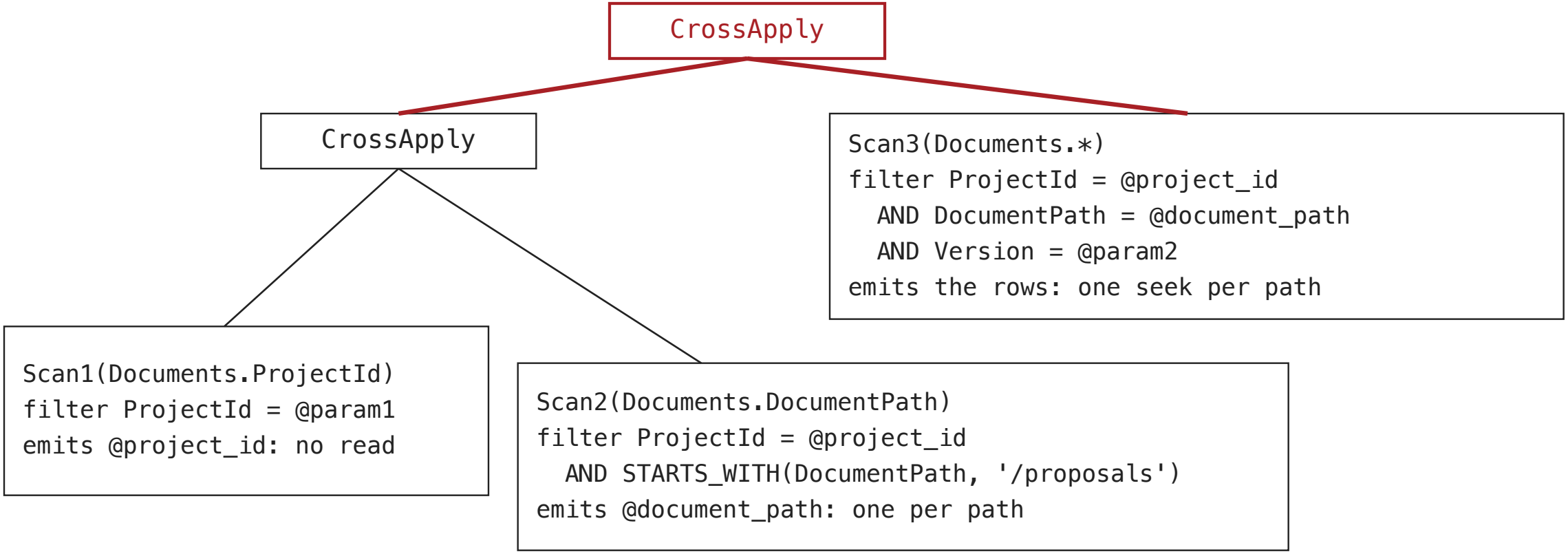
Pin one key column, then constrain the next given its value: one scan per key column.



Scan3 handles Version, but needs a `@document_path` from somewhere.

Key columns one at a time

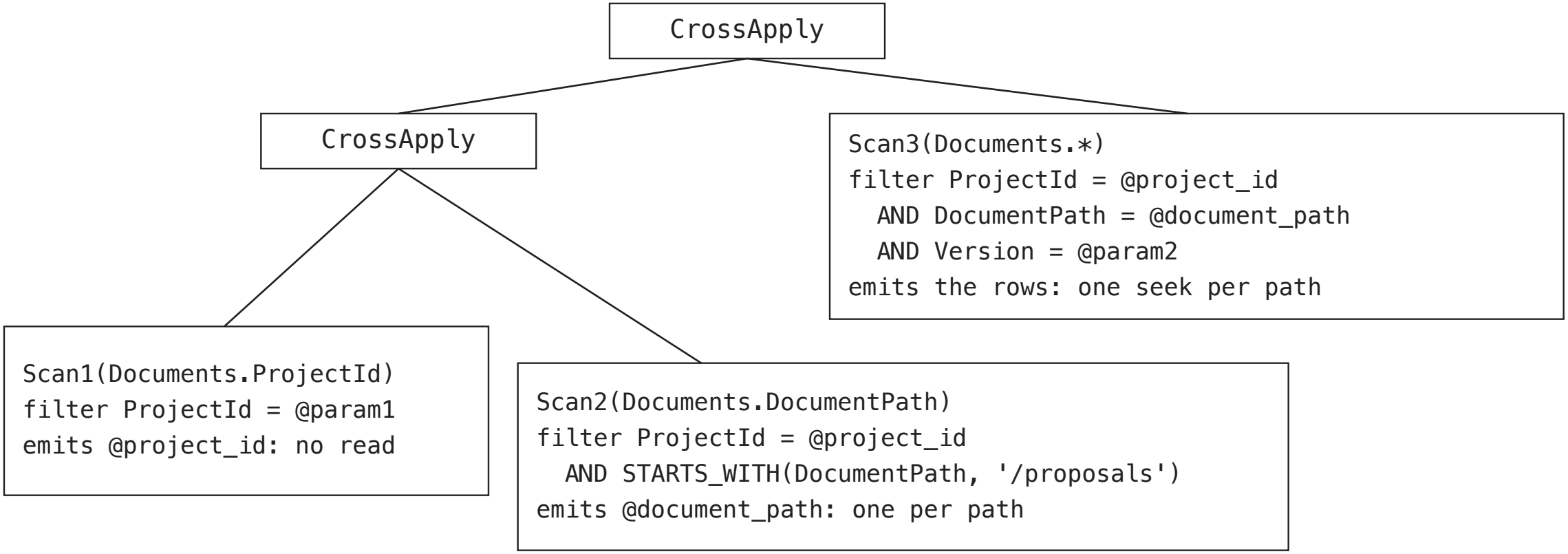
Pin one key column, then constrain the next given its value: one scan per key column.



For each `@document_path`, run Scan3: one seek to its 2017-01-01 version. Ten rows.

Key columns one at a time

Pin one key column, then constrain the next given its value: one scan per key column.



Scan1 pins the project; Scan2 lists its paths; Scan3 seeks each path's version.

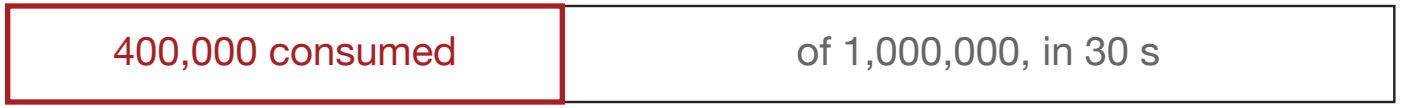
Restarts

Failures in the middle of a query

one long query

50 shards, one timestamp

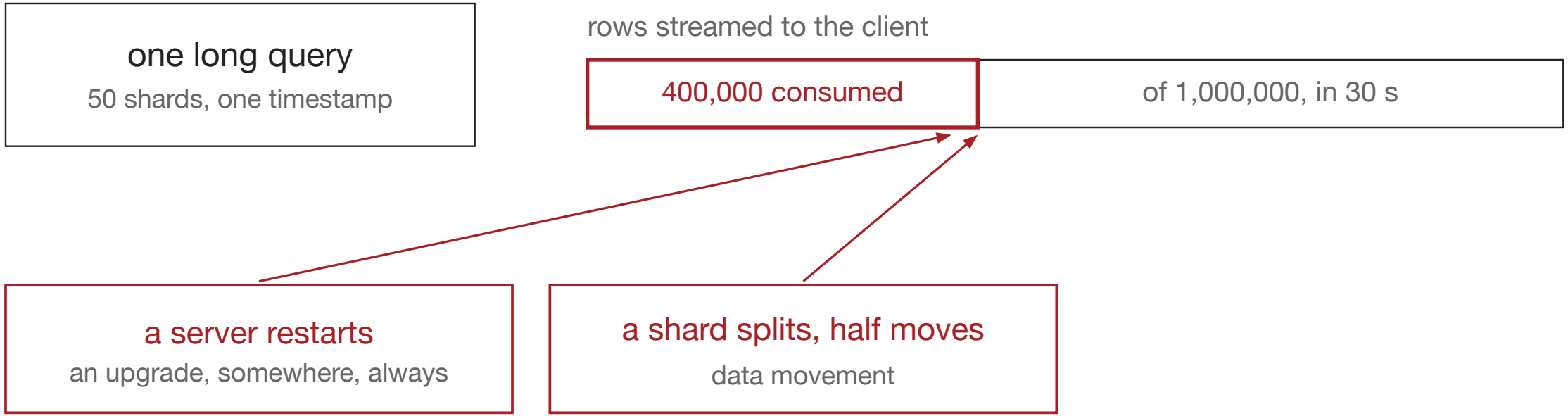
rows streamed to the client



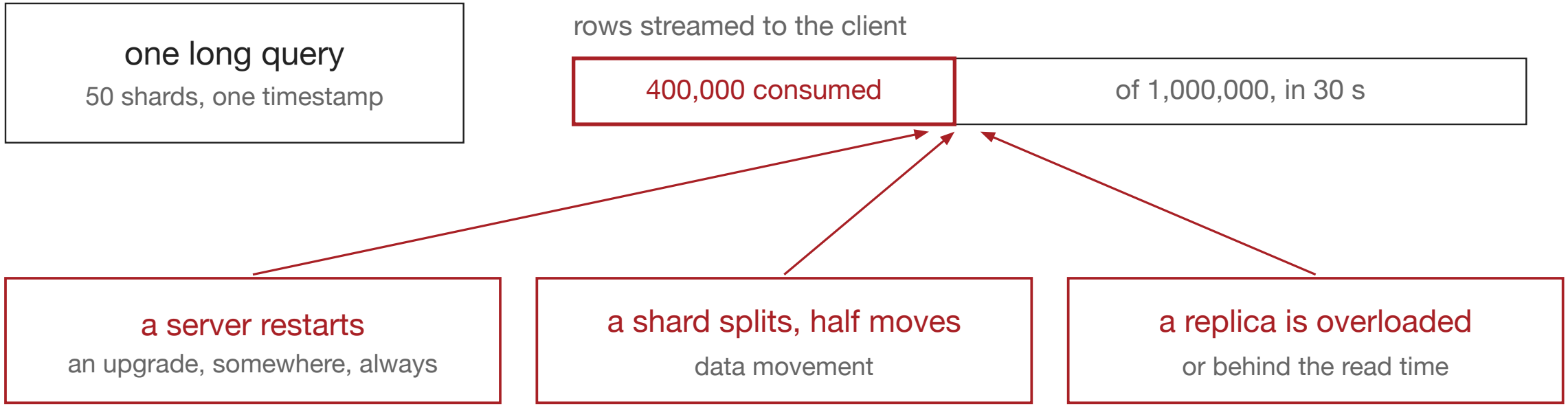
Failures in the middle of a query



Failures in the middle of a query



Failures in the middle of a query



Something is always upgrading, moving or overloaded.

Three strawmen

Strawman	What it costs
Return an error; the client reruns the query	
Make the results durable	
Rewrite the query to resume: row numbers (identity columns), or pages (key > last)	

Three strawmen

Strawman	What it costs
Return an error; the client reruns the query	retry loops in every application; a long query may never finish; one bad shard sets the latency of fifty
Make the results durable	
Rewrite the query to resume: row numbers (identity columns), or pages (key > last)	

Three strawmen

Strawman	What it costs
Return an error; the client reruns the query	retry loops in every application; a long query may never finish; one bad shard sets the latency of fifty
Make the results durable	every query pays latency to protect the few that fail
Rewrite the query to resume: row numbers (identity columns), or pages (key > last)	

Three strawmen

Strawman	What it costs
Return an error; the client reruns the query	retry loops in every application; a long query may never finish; one bad shard sets the latency of fifty
Make the results durable	every query pays latency to protect the few that fail
Rewrite the query to resume: row numbers (identity columns), or pages (key > last)	identity columns for every intermediate row: no general method found; paging adds a sort

Restart tokens

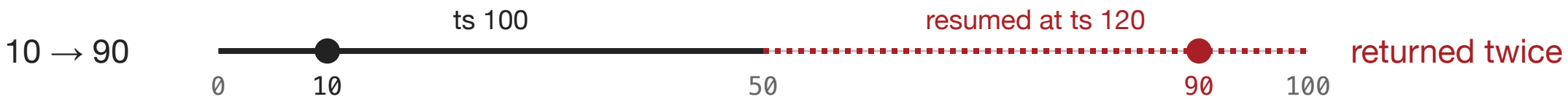
- Every batch of results carries an opaque **restart token**. To resume, the client library resends the request with the last token it got.
- Contract: the rows before and after a restart form a correct answer, none twice; not necessarily the one an unbroken run would give.
- A range scan’s token: the last key returned and the **snapshot timestamp**, the one time all the query’s reads use. Resume: seek just past that key, at that timestamp.

Nothing is written to storage for the sake of restarts.

`SELECT * FROM T LIMIT 100`, restarted after 30 rows: 70 more, none repeated, maybe not the first run’s 70.

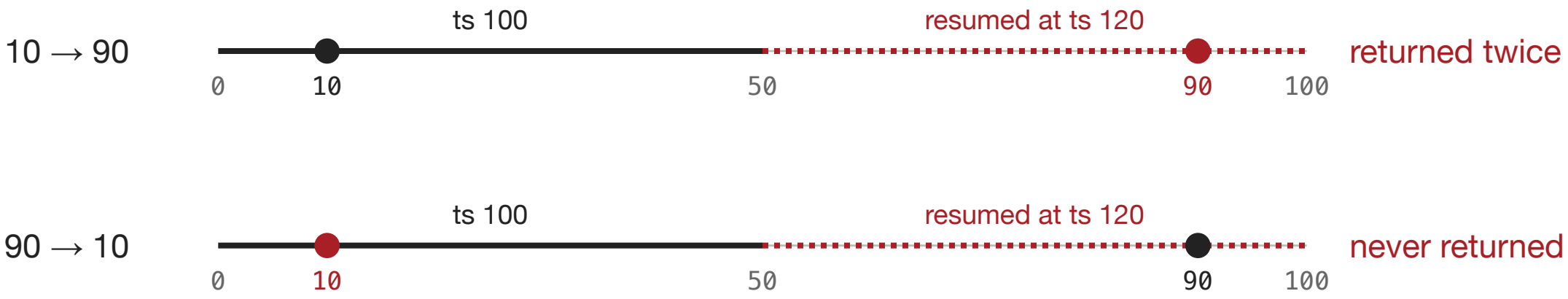
Why the token carries the timestamp

At timestamp (ts) 110, row r's key changes, between the two halves of one scan:



Why the token carries the timestamp

At timestamp (ts) 110, row r's key changes, between the two halves of one scan:



Resume at the token's timestamp: both halves read one snapshot.

Any replica whose `t_safe`, the time up to which it has applied every write, has passed that timestamp returns the same data, even in another datacenter.

The limit: old versions are garbage-collected, and that error still reaches the client.

Not for read-write transactions

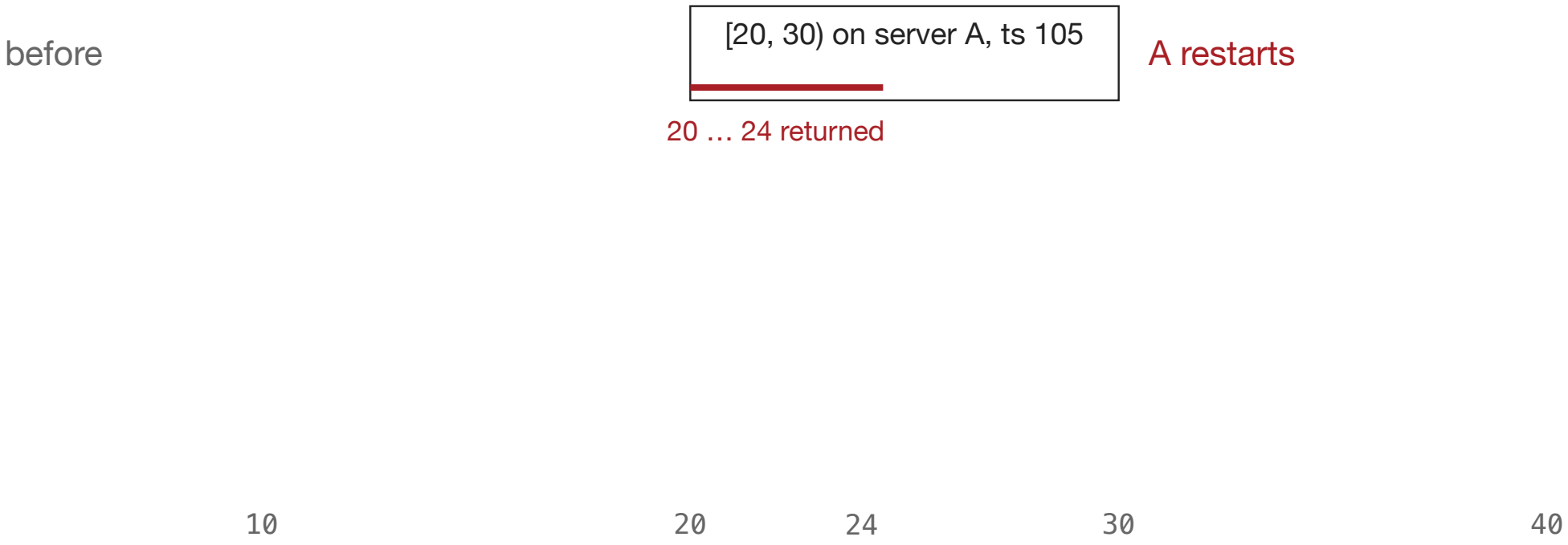
- Snapshot queries, strong ones (read at the current time) included, are most of Spanner's queries; their timestamp is fixed once.
- A read-write transaction reads the latest data,
  valid only while its locks are held.
- Those locks live only in the leader replica's memory. A failure often loses them, and another transaction may change rows already read.

No snapshot to resume into: the transaction aborts, and the client retries.

As it already must after a wound-wait abort, Oct 19's deadlock-avoidance rule.

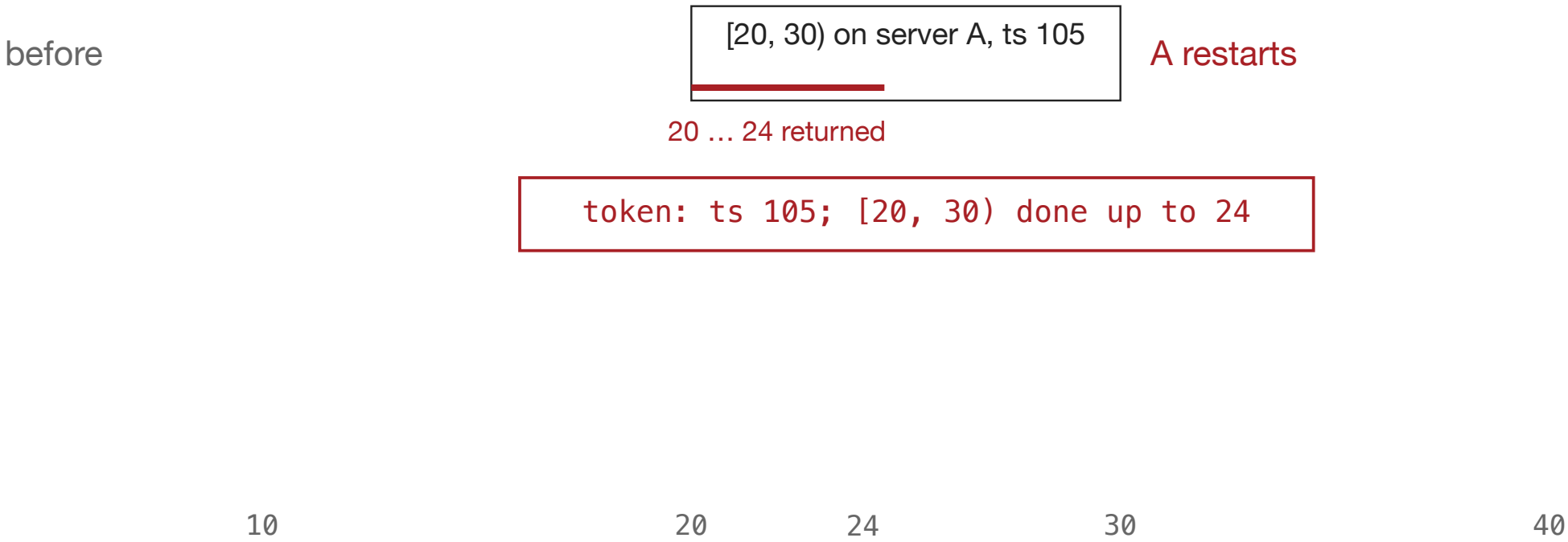
What a token must survive: resharding

The server that resumes may own a different key range from the one that stopped.



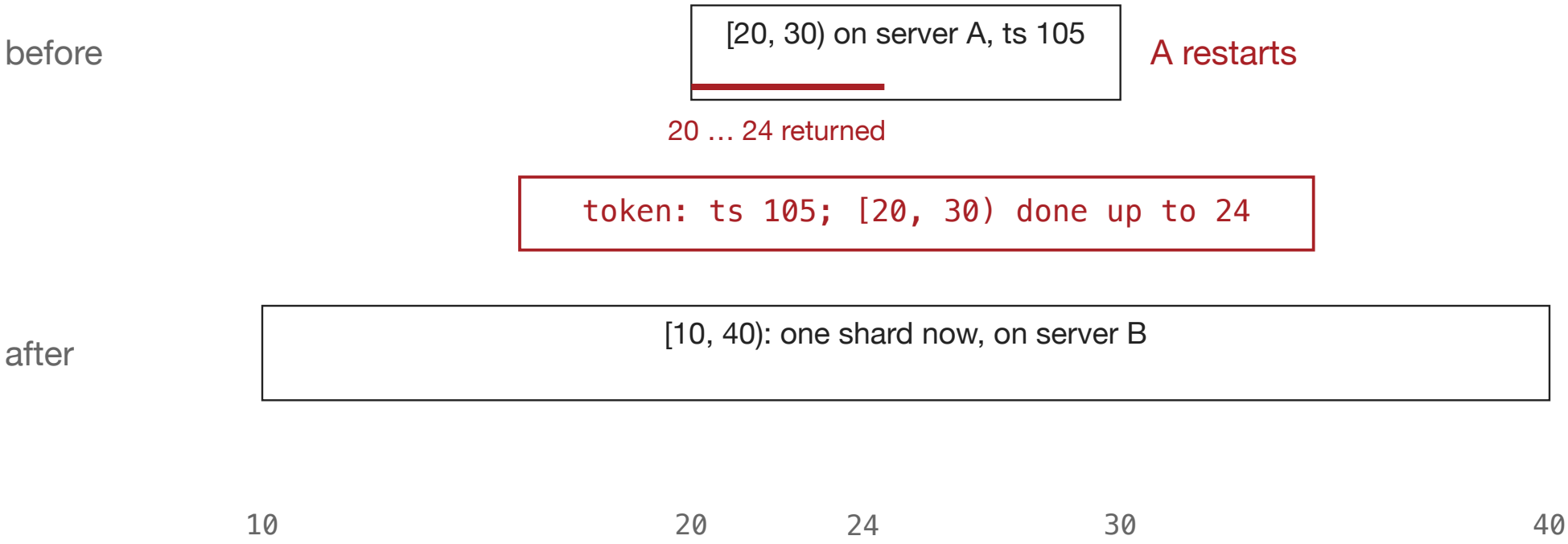
What a token must survive: resharding

The server that resumes may own a different key range from the one that stopped.



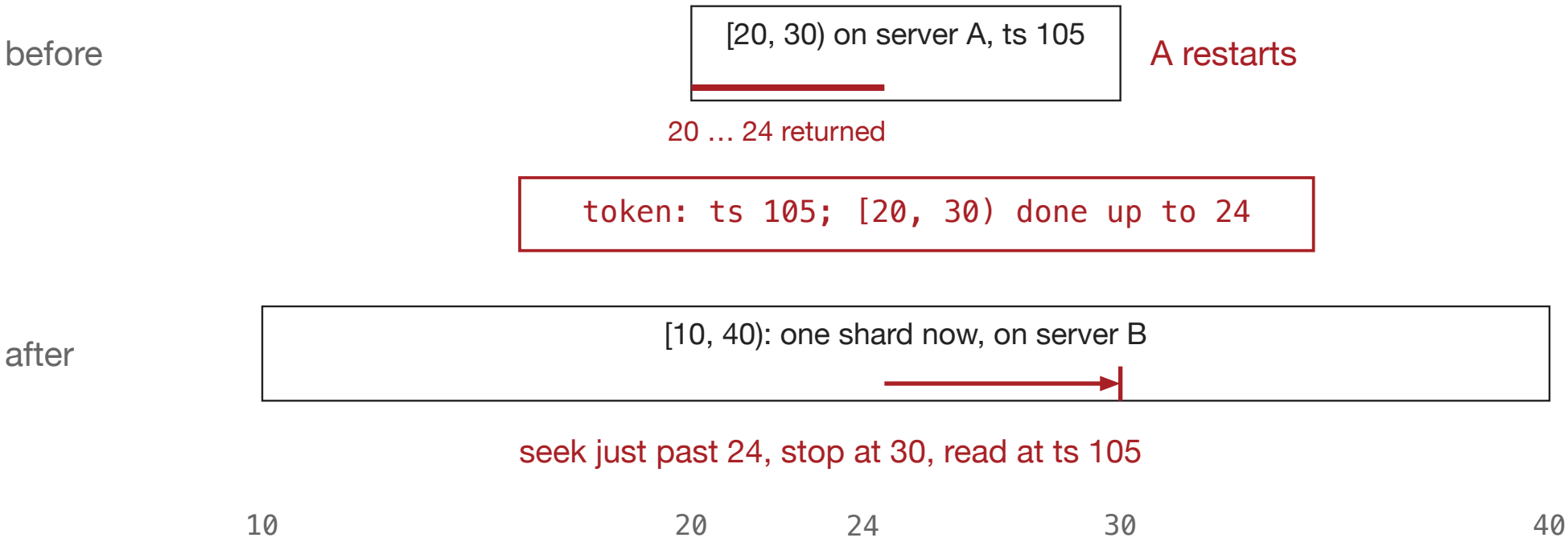
What a token must survive: resharding

The server that resumes may own a different key range from the one that stopped.



What a token must survive: resharding

The server that resumes may own a different key range from the one that stopped.

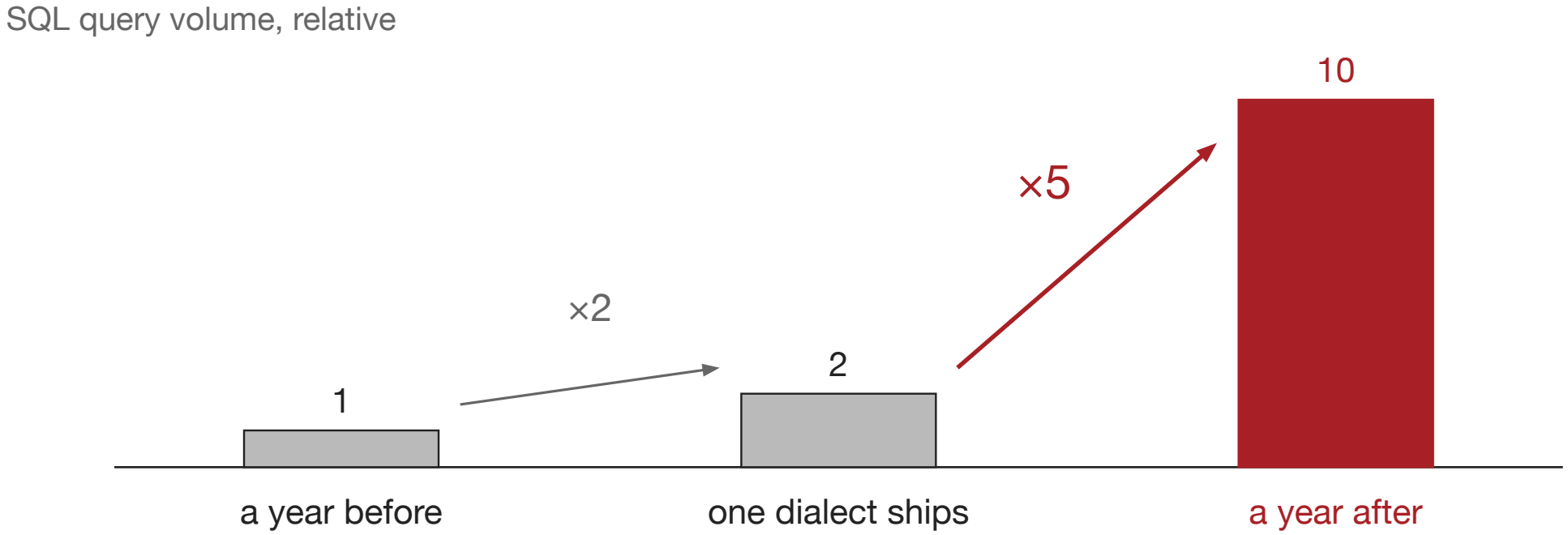


“Shard 3, 400 rows done” means nothing on B: record progress in key space.

Spanner Lessons

Users wanted SQL

Key-value lookups and scans got Spanner launched; anything more complex was easier in SQL.

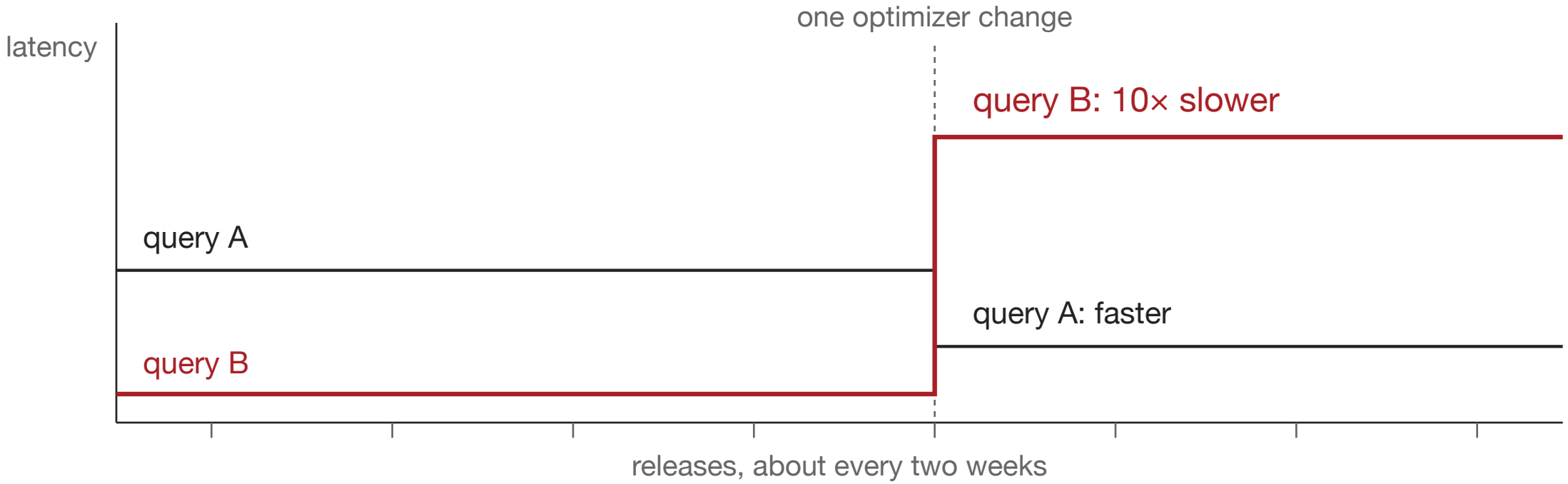


With one dialect across Google, SQL use grew fivefold in a year.

Prospective Cloud customers asked for SQL too.

Plans must not regress

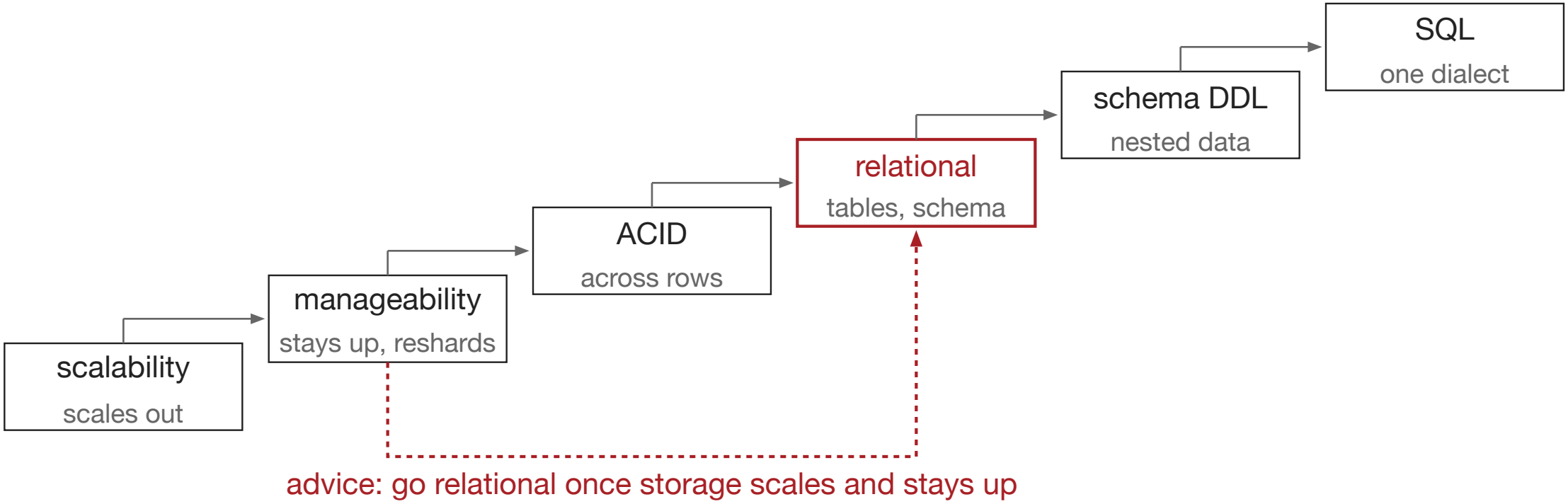
A new release about every two weeks, under systems that must never get slower.



Improving the optimizer makes some plans worse.

A changed plan can swing latency by an order of magnitude. The curves are illustrative.

Scale first, then go relational

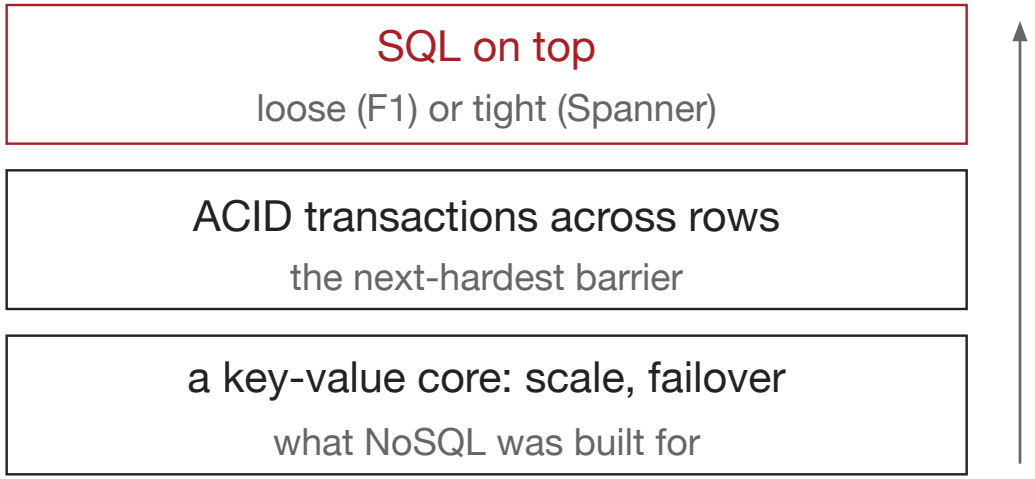


Once storage scales and stays up, adopt the relational model early.

Relational abstractions speed development and make later migrations cheaper.

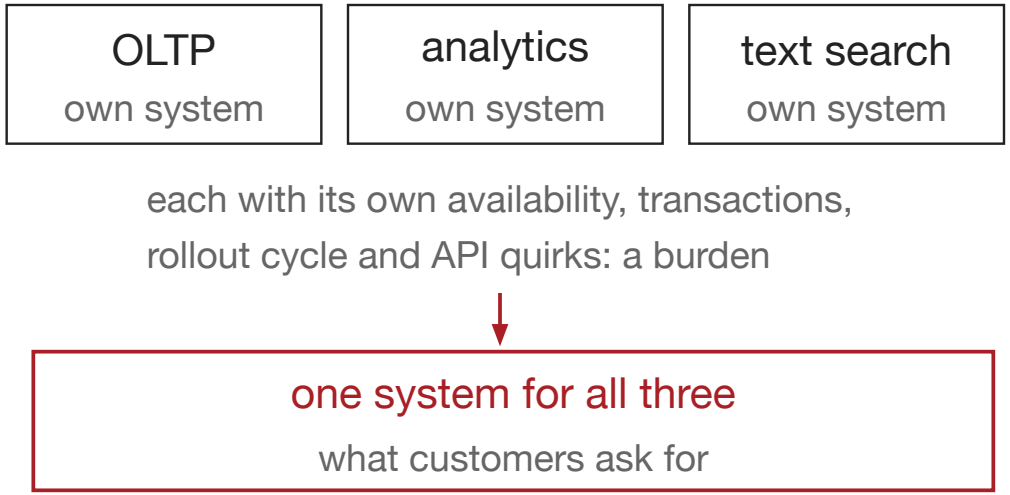
SQL vs. NoSQL, and one size fits all

SQL vs. NoSQL



built bottom up: a stack, not a choice

One size fits all?



the goal: do well across that range, over time

Recap

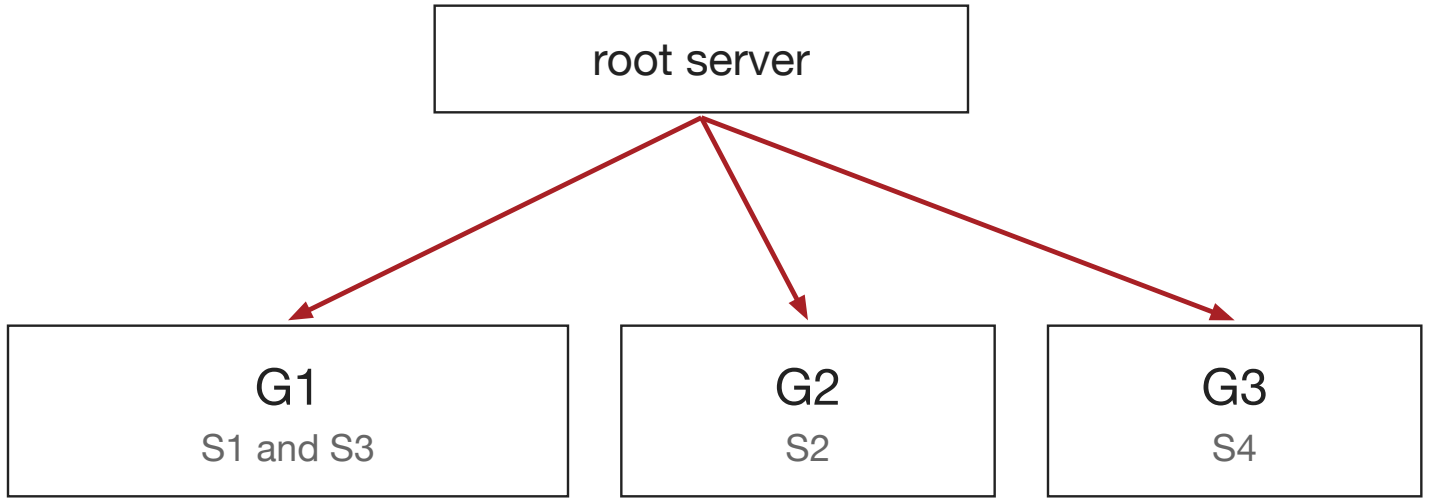
1. **Distribution** DistributedUnion · pull up · partitionability · split Top
2. **Locality** interleaving · DistributedApply · batches
3. **Run time** shard pruning
4. **Ranges** one scan per key column · seeks
5. **Restarts** restart token · timestamp · key space
6. **Lessons** SQL · stable plans · scale first · one system

Backup

Which servers: fewer calls

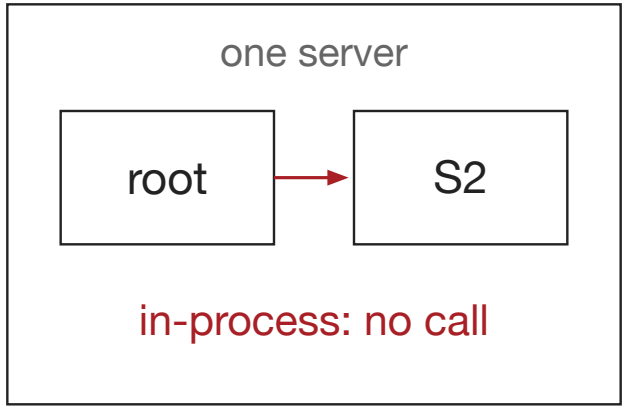
When the ranges cover the whole table, call each Paxos group once, in parallel.

All four shards: one call per group



4 shards, 3 calls

Shard affinity

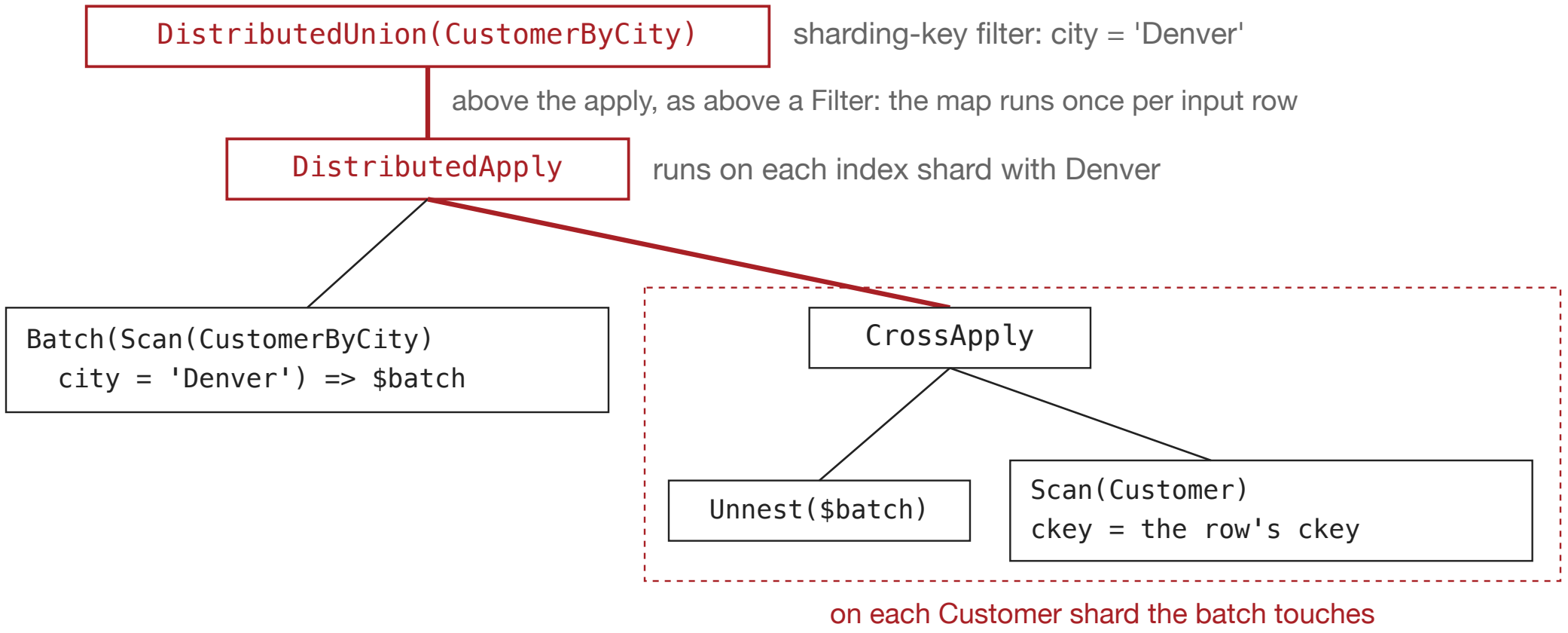


The shard is here, so the subtree runs here.

Count calls, not shards.

Affinity is common: many queries read one object, a user and its interleaved rows, under one key.

Q2, distributed



Fits a small input joined on the other table's sharding key: index to base table, foreign key to parent.

Does not fit two big tables joined on a non-key column: every batch goes to every shard.

Send the query to its data

A root server that holds none of the data costs an extra hop:

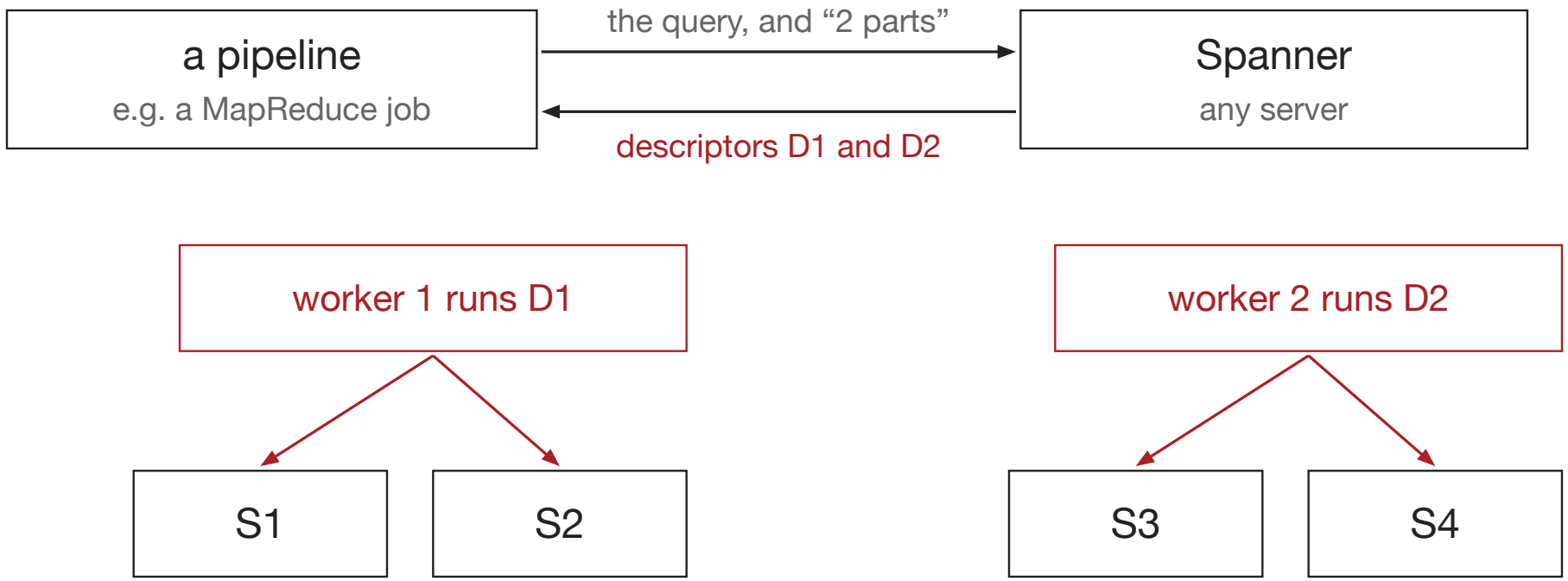
two cross-machine calls where one would do.

```
SELECT c.*, s.* FROM Customer c JOIN Sales s ON c.ckey = s.ckey WHERE c.ckey = @k
  hint: Customer key = @k          → the server with customer @k; Sales is
                                     interleaved, so the whole query runs there
SELECT c.* FROM Customer c WHERE c.city = @city
  hint: CustomerByCity key = @city → the server with that index shard
```

After the first run, the client knows where to send the query.

A **location hint**: the outermost union's sharding-key filter, in terms of the parameters, cached on the client.

Too big for one client



Each worker calls its shards directly; all read at one timestamp.

Only for **root-partitionable** queries: the plan’s root is a DistributedUnion.

The workers are **parallel consumers**. Q1 does not qualify: its final Top(5) sits above the union.

Locking rows that are not there yet

- Spanner locks per column. Key columns of a range stand for which rows exist there: an insert or delete in the range conflicts.
- A non-key column of a row stands for its value: an update of that row conflicts.
- The answer depends on which documents are **absent**. `/proposals/doc11` is not a row yet, so lock the range its insert would land in.

A row appearing in a range already read is a phantom: Nov 11.

Snapshot reads, lock-free reads at one timestamp, lock nothing; for them the ranges say which prepared but uncommitted writes to wait out.