Lessons in Building a Distributed Query Planner

Ozgun Erdogan

PGCon 2016

citusdata

Talk Outline

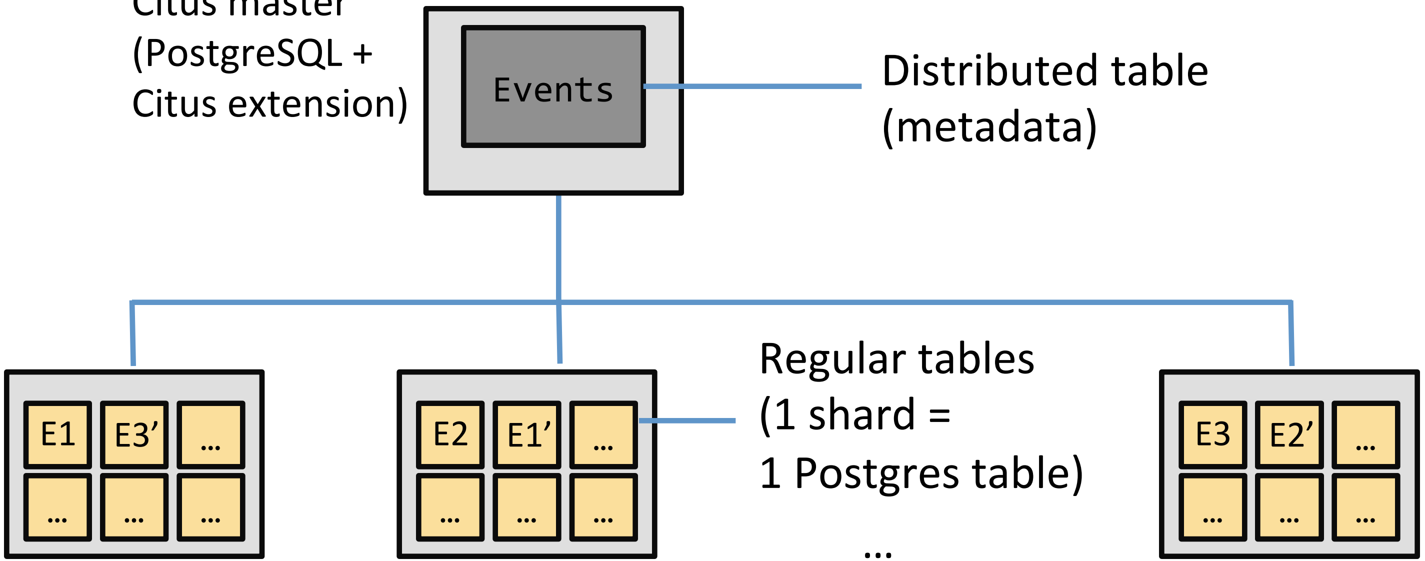
1. Introduction
2. Key insight in distributed planning
3. Distributed logical plans
4. Distributed physical plans
5. Different workloads: Different executors

- Four technical lightning talks in one

What is Citus?

- Citus extends PostgreSQL (not a fork) to provide it with distributed functionality.
- Citus scales-out Postgres across servers using sharding and replication. Its query engine parallelizes SQL queries across many servers.
- Citus 5.0 is open source: https://github.com/citusdata/citus

Citus 5.0 Architecture Diagram



Citus master
(PostgreSQL +
Citus extension)

Distributed table
(metadata)

Regular tables
(1 shard =
1 Postgres table)

…

Citus worker 1
(PostgreSQL +
Citus extension)

Citus worker 2

Citus worker N

citusdata

When is Citus a good fit?

- Sub-second OLAP queries on data as it arrives
  - Powering real-time analytic dashboards
  - Exploratory queries on events as they arrive
- Who is using Citus?
  - CloudFlare uses Citus to power their analytic dashboards
  - Neustar builds ad-tech infrastructure with HyperLogLog
  - Heap powers funnel, segmentation, and cohort queries
- Citus *isn't* a good fit to replace your data warehouse.

Why is distributed query planning (SELECTs) hard?

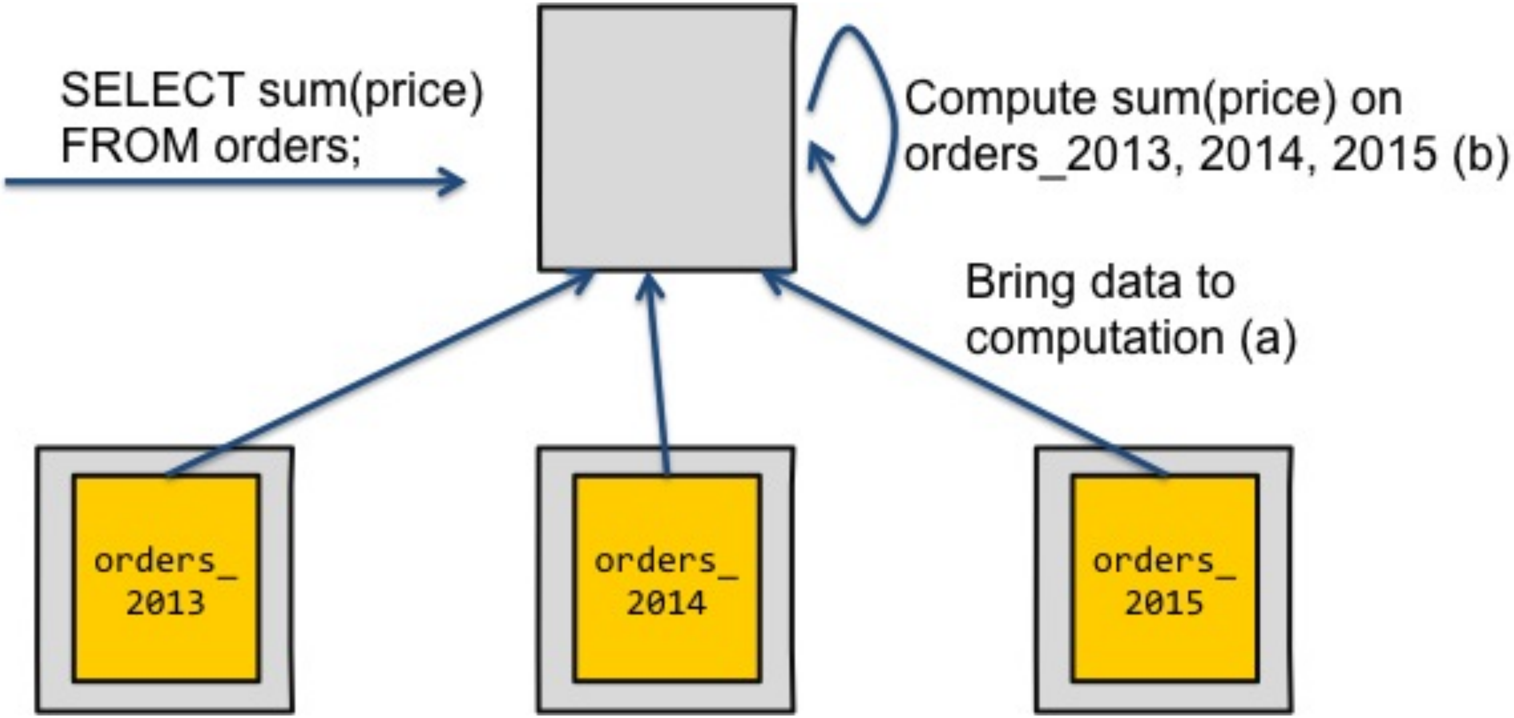
Past Experiences

- Built a similar distributed data processing engine at Amazon called CSPIT
- Led by a visionary architect and built by an extremely talented team
- Scaled to (at best) a dozen machines. Nicely distributed basic computations across machines
  - Then the dream met reality

Why did it fail?

- You can solve all distributed systems problems in one of two days:
  1. Bring your data to the computation
  2. Push your computation to the data

Bringing data to computation (1)

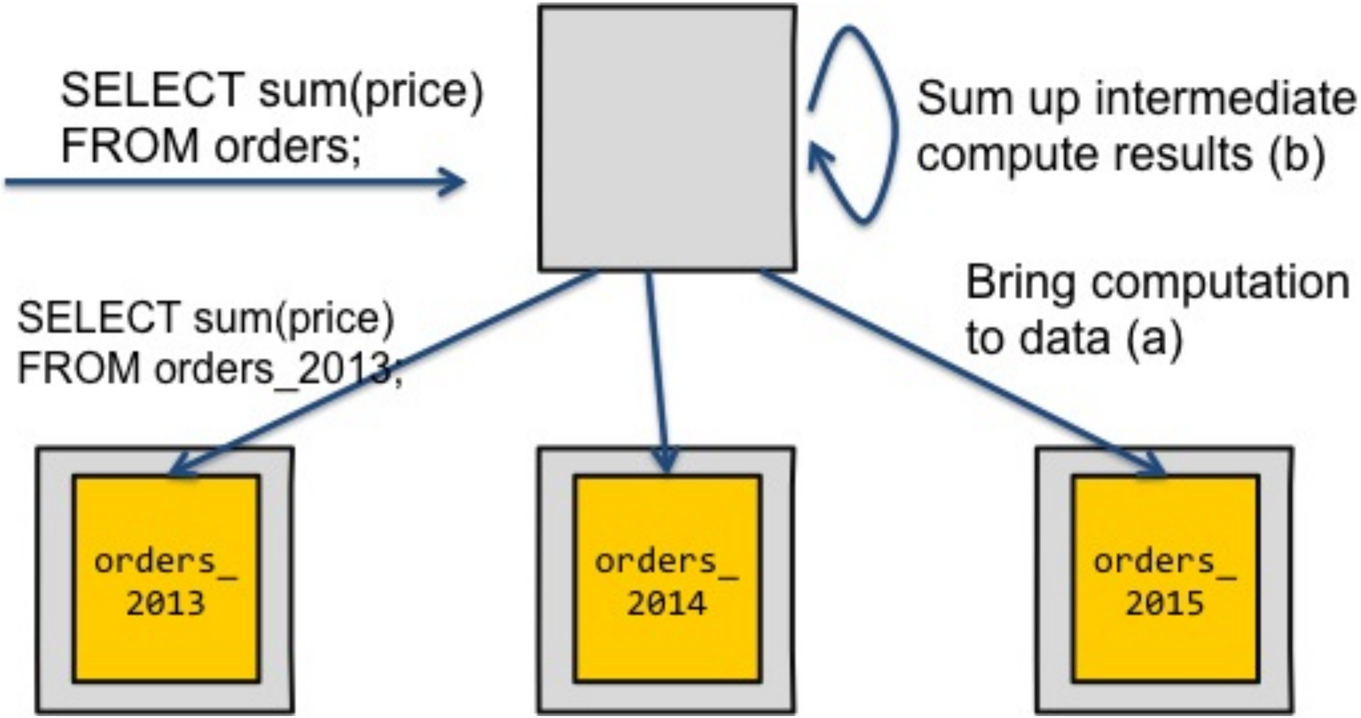


SELECT sum(price)
FROM orders;

Compute sum(price) on orders_2013, 2014, 2015 (b)

Bring data to computation (a)

Bringing computation to data (2)



Slightly more complex queries

- Sum(price): sum(price) on worker nodes and then sum() intermediate results
- Avg(price): Can you avg(price) on worker nodes and then avg() intermediate results?
  - Why not?

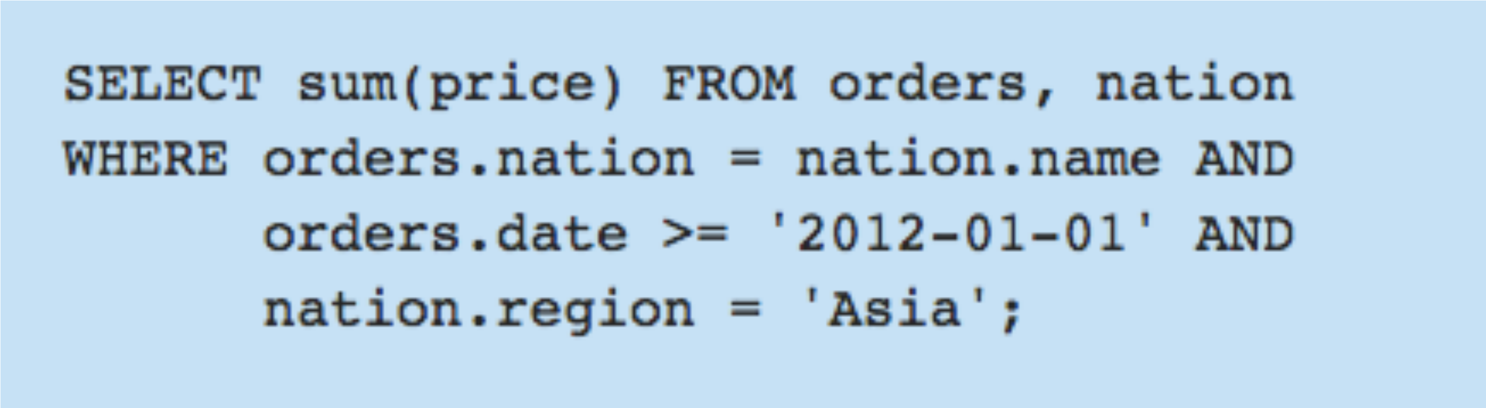
Commutative Computations

- If you can transform your computations into their commutative form, then you can push them down.
  - (a + b = b + a ; a / b ≠ b / a) (*)
- Associative and distributive property for other operations (We also knew about this)

How does this help me?

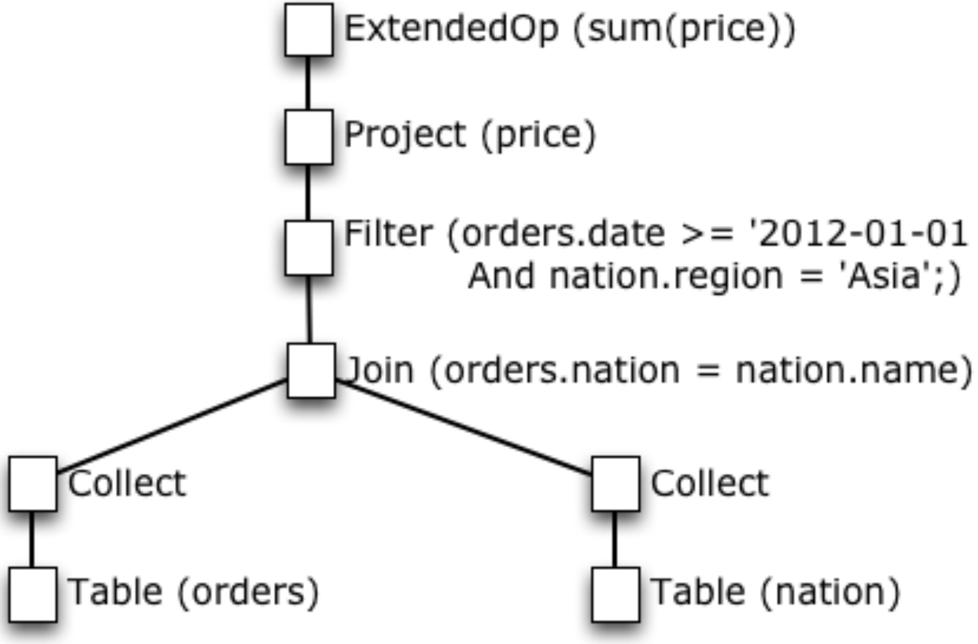
- Commutative, associative, and distributive properties hold for any query language
- We pick SQL as an example language
- SQL uses Relational Algebra to express a query
- If a query has a WHERE clause in it, that’s a FILTER node in the relational algebra tree

Simple SQL query



```
SELECT sum(price) FROM orders, nation
WHERE orders.nation = nation.name AND
      orders.date >= '2012-01-01' AND
      nation.region = 'Asia';
```

Distributed Logical Plan (unoptimized)



ExtendedOp (sum(price))

Project (price)

Filter (orders.date >= '2012-01-01
And nation.region = 'Asia';)

Join (orders.nation = nation.name)

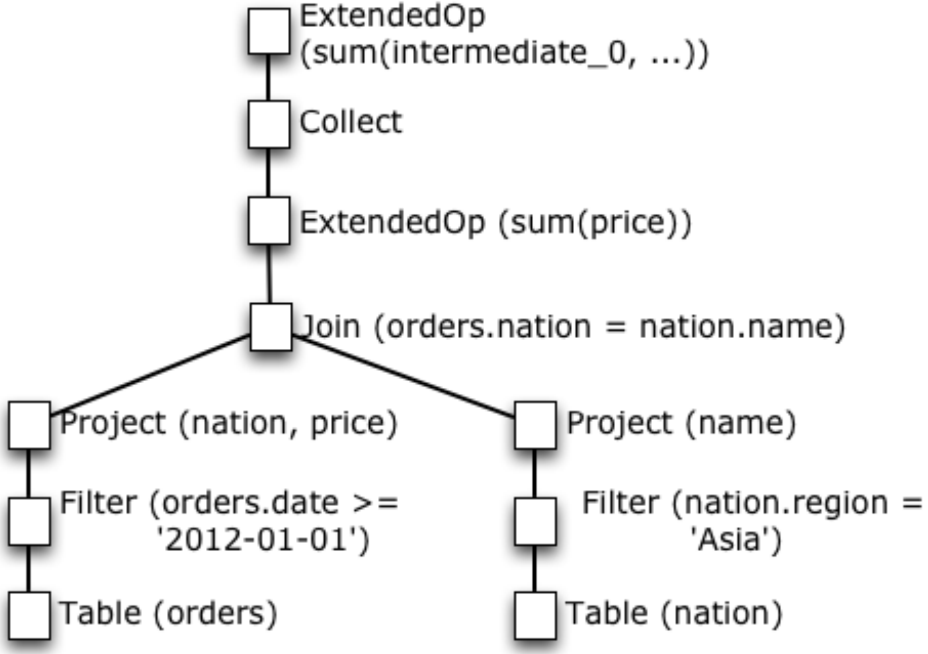
Collect

Table (orders)

Collect

Table (nation)

Distributed Logical Plan (optimized)



ExtendedOp
(sum(intermediate_0, ...))

Collect

ExtendedOp (sum(price))

Join (orders.nation = nation.name)

Project (nation, price)

Filter (orders.date >=
'2012-01-01')

Table (orders)

Project (name)

Filter (nation.region =
'Asia')

Table (nation)

Takeaway

In the land of distributed systems, the commutative (and distributive) property is king! Transform your queries with respect to the king, and your network I/O will scale.

From Example to Distributed Logical Plans

One example doesn’t make a proof

- Can you prove this model is complete?
- Relational Algebra has 10 operators
- What about optimizing more complex plans with joins, subselects, and other constructs?

Multi-Relational Algebra

- Correctness of Query Execution Strategies in Distributed Databases Ceri and Pelagatti, 1983
  - A Distributed Database paper from a more civilized age
- Models each relational algebra operator as a distributed operator and extends it

Collect and Repartition Operators

- Collect operator merges data underneath in one place
- Repartition operator takes a “relation” partitioned on one dimension, and repartitions it on a different dimension

Commutative Property Rules

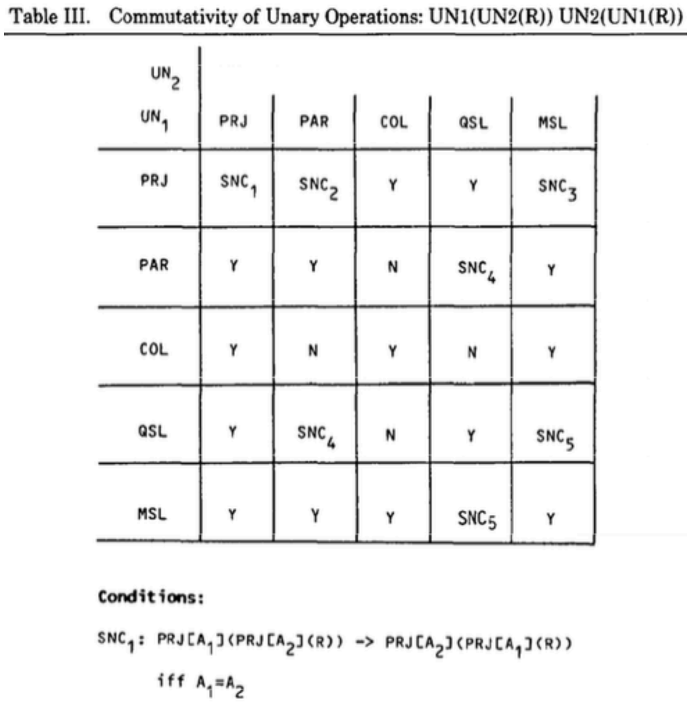


Table III. Commutativity of Unary Operations: UN1(UN2(R)) UN2(UN1(R))

UN_2 UN_1	PRJ	PAR	COL	QSL	MSL
PRJ	SNC_1	SNC_2	Y	Y	SNC_3
PAR	Y	Y	N	SNC_4	Y
COL	Y	N	Y	N	Y
QSL	Y	SNC_4	N	Y	SNC_5
MSL	Y	Y	Y	SNC_5	Y

Conditions:

SNC_1: $PRJ[A_1](PRJ[A_2](R)) \to PRJ[A_2](PRJ[A_1](R))$

iff $A_1=A_2$

Distributive Property Rules

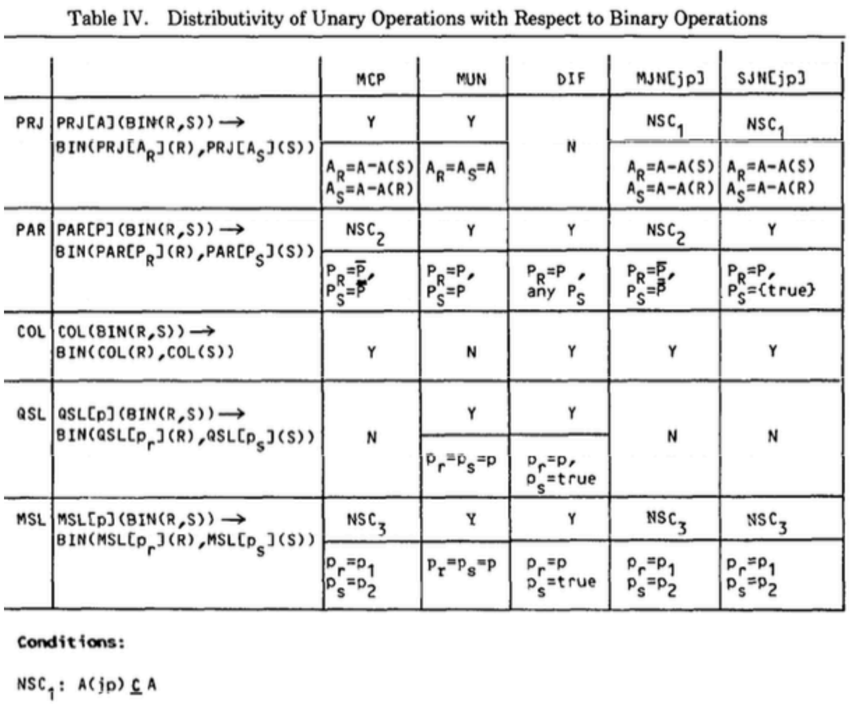


Table IV. Distributivity of Unary Operations with Respect to Binary Operations

		MCP	MUN	DIF	MJN[jp]	SJN[jp]
PRJ	PRJ[A](BIN(R,S)) → BIN(PRJ[A_R](R),PRJ[A_S](S))	Y	Y	N	NSC_1	NSC_1
		$A_R=A-A(S)$ $A_S=A-A(R)$	$A_R=A_S=A$		$A_R=A-A(S)$ $A_S=A-A(R)$	$A_R=A-A(S)$ $A_S=A-A(R)$
PAR	PAR[P](BIN(R,S)) → BIN(PAR[P_R](R),PAR[P_S](S))	NSC_2	Y	Y	NSC_2	Y
		$P_R=\bar{P}$, $P_S=\bar{P}$	$P_R=P$, $P_S=P$	$P_R=P$, any P_S	$P_R=\bar{P}$, $P_S=\bar{P}$	$P_R=P$, $P_S=\{true\}$
COL	COL(BIN(R,S)) → BIN(COL(R),COL(S))	Y	N	Y	Y	Y
QSL	QSL[p](BIN(R,S)) → BIN(QSL[p_r](R),QSL[p_s](S))	N	Y	Y	N	N
			$p_r=p_s=p$	$p_r=p$, $p_s=true$		
MSL	MSL[p](BIN(R,S)) → BIN(MSL[p_r](R),MSL[p_s](S))	NSC_3	Y	Y	NSC_3	NSC_3
		$p_r=p_1$ $p_s=p_2$	$p_r=p_s=p$	$p_r=p$ $p_s=true$	$p_r=p_1$ $p_s=p_2$	$p_r=p_1$ $p_s=p_2$

Conditions:

NSC_1: $A(jp) \subseteq A$

Factorization Rules

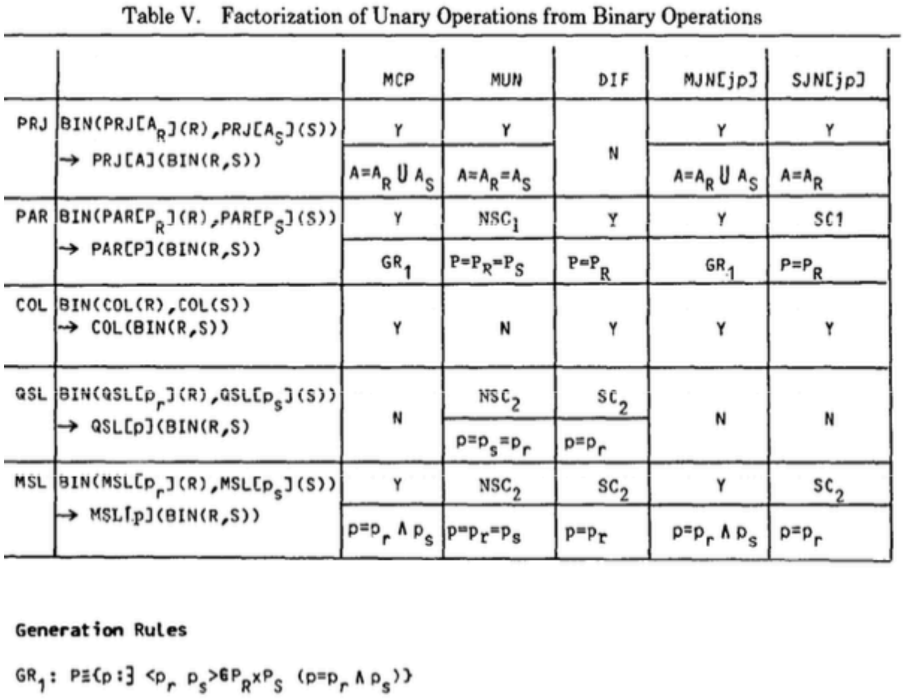


Table V. Factorization of Unary Operations from Binary Operations

		MCP	MUN	DIF	MJN[jp]	SJN[jp]
PRJ	BIN(PRJ[A_R](R),PRJ[A_S](S))	Y	Y	N	Y	Y
	→ PRJ[A](BIN(R,S))	$A=A_R \cup A_S$	$A=A_R=A_S$		$A=A_R \cup A_S$	$A=A_R$
PAR	BIN(PAR[P_R](R),PAR[P_S](S))	Y	NSC_1	Y	Y	SC1
	→ PAR[P](BIN(R,S))	GR_1	$P=P_R=P_S$	$P=P_R$	GR_1	$P=P_R$
COL	BIN(COL(R),COL(S)) → COL(BIN(R,S))	Y	N	Y	Y	Y
QSL	BIN(QSL[p_r](R),QSL[p_s](S))	N	NSC_2	SC_2	N	N
	→ QSL[p](BIN(R,S)		$p=p_s=p_r$	$p=p_r$		
MSL	BIN(MSL[p_r](R),MSL[p_s](S))	Y	NSC_2	SC_2	Y	SC_2
	→ MSL[p](BIN(R,S))	$p=p_r \wedge p_s$	$p=p_r=p_s$	$p=p_r$	$p=p_r \wedge p_s$	$p=p_r$

Generation Rules

GR_1: $P \equiv \{p : \exists \langle p_r\ p_s \rangle \in P_R \times P_S\ (p=p_r \wedge p_s)\}$

Takeaway

Multi-relational Algebra (MRA) offers a complete foundation for distributing SQL queries.

Note: Citus is adding more SQL functionality with each release. Citus works best when you need to ingest and query large volumes of data in human real-time.

citusdata

From Distributed Logical to Distributed Physical Plan

Logical plan ≠ Physical plan

- “Table” is a logical operator. SequentialScan or BitmapIndexScan is a physical operator.
- “Join” is a logical operator. HashJoin or MergeJoin is a physical operator.
- Distributed databases that start with a database usually just add physical operators. (Greenplum, Redshift)

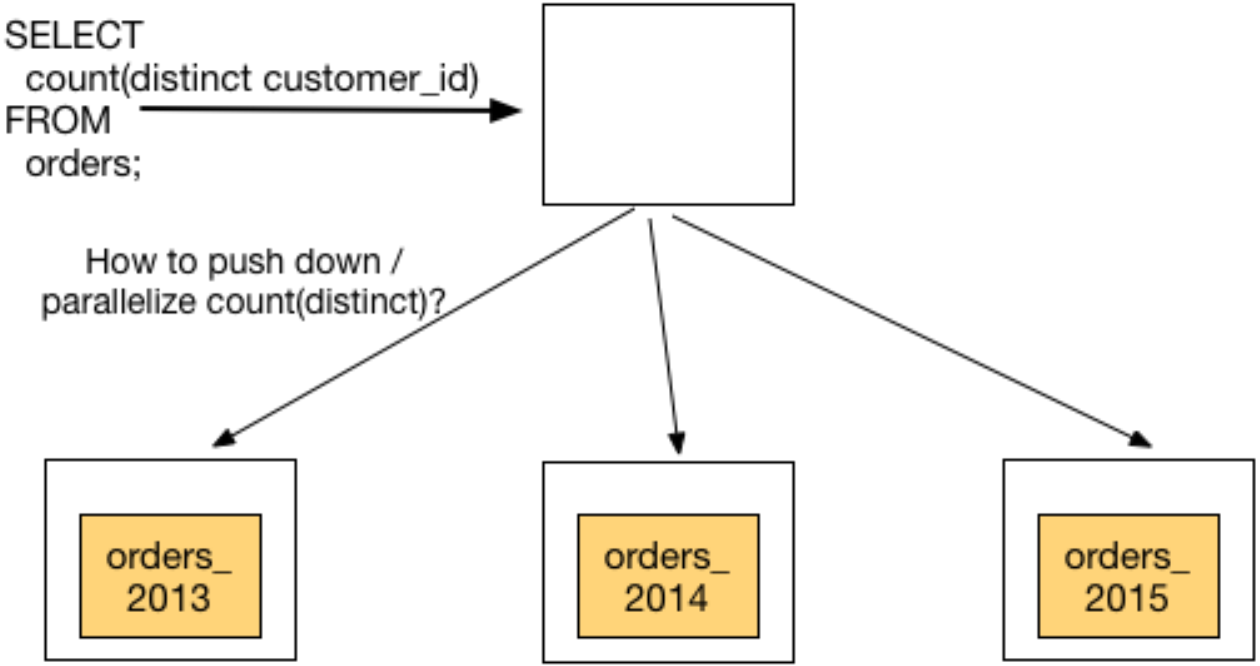
Logical to Physical Plans

- *If* you have a distributed logical plan, you can map that to a physical plan in different ways.
- Multi-relational Algebra defines relational algebra operators, Collect, and Repartition
  1. All standard operators -> SQL
  2. Collect -> Copy data
  3. Repartition -> Map/Reduce

SQL as a physical operator

- Defining “SQL” as an execution primitive decouples local execution internals from distributed execution.
  1. Decouple network and disk I/O related planning. Delegate disk I/O optimizations to PostgreSQL
  2. Automatically pick up improvements in Postgres. Also benefit from LLVM and vectorized execution

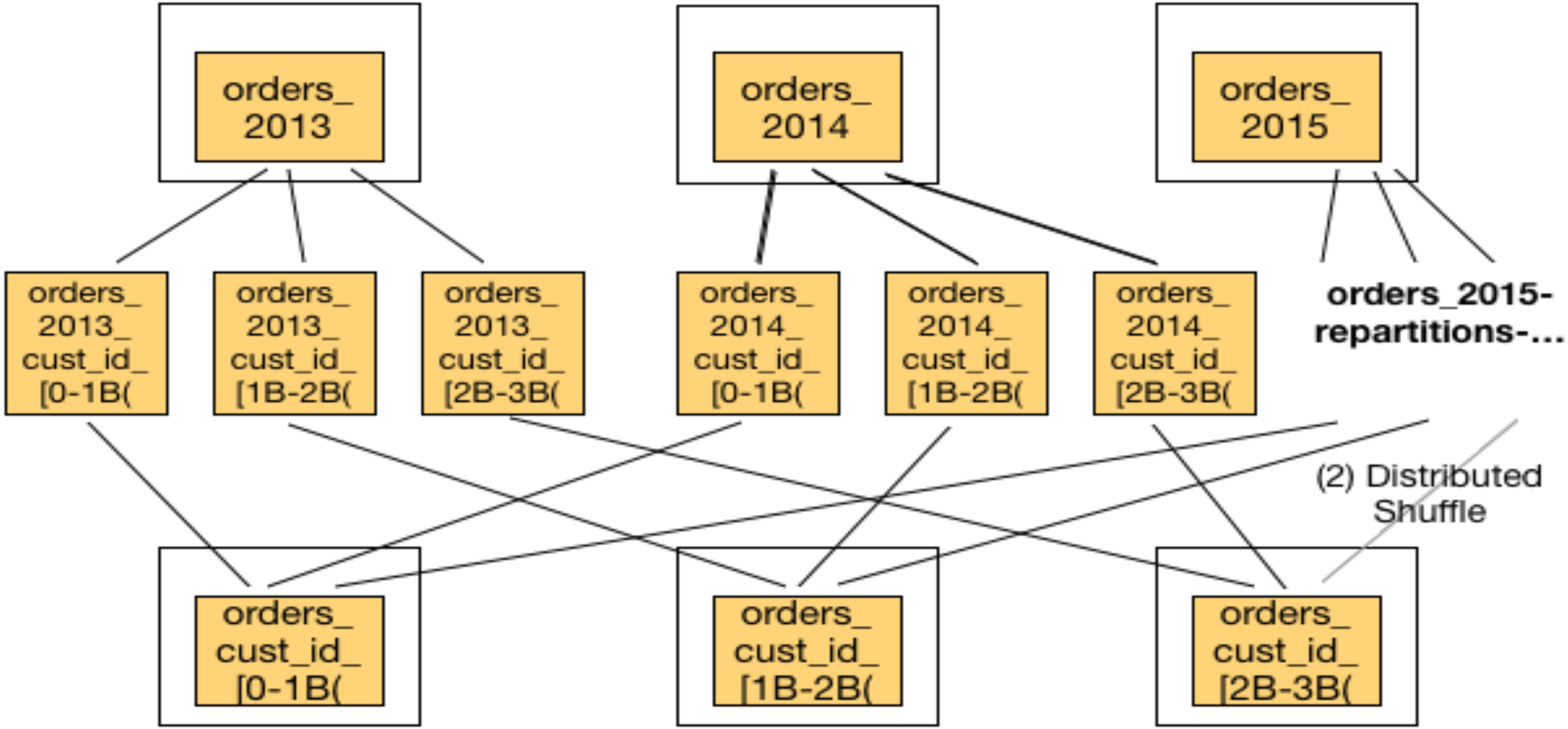
Repartition through an example (1)



SELECT
  count(distinct customer_id)
FROM
  orders;

How to push down /
parallelize count(distinct)?

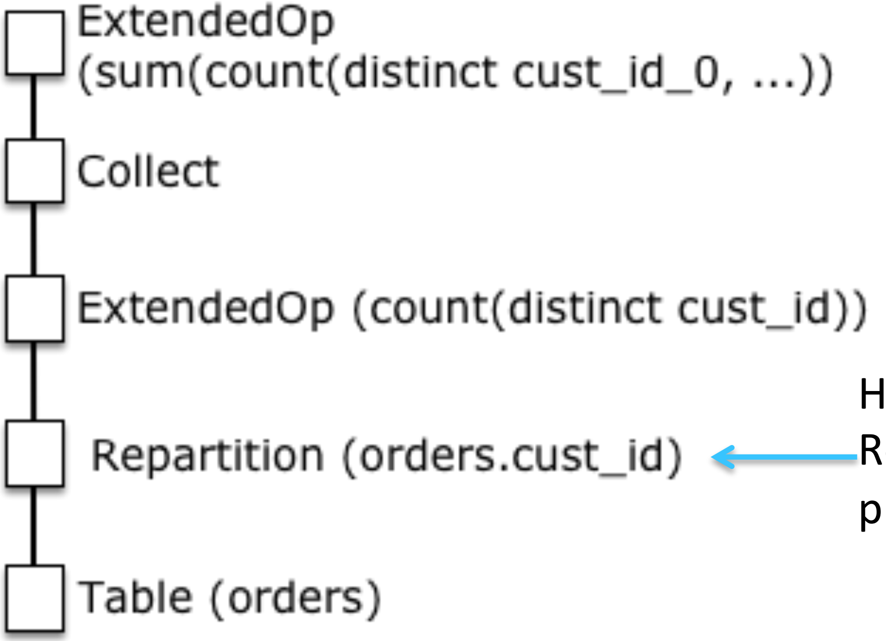
Repartition through an example (2)



citusdata

Repartition in Logical Plan

```
SELECT
  count(distinct cust_id)
FROM
  orders;
```



ExtendedOp
(sum(count(distinct cust_id_0, ...))

Collect

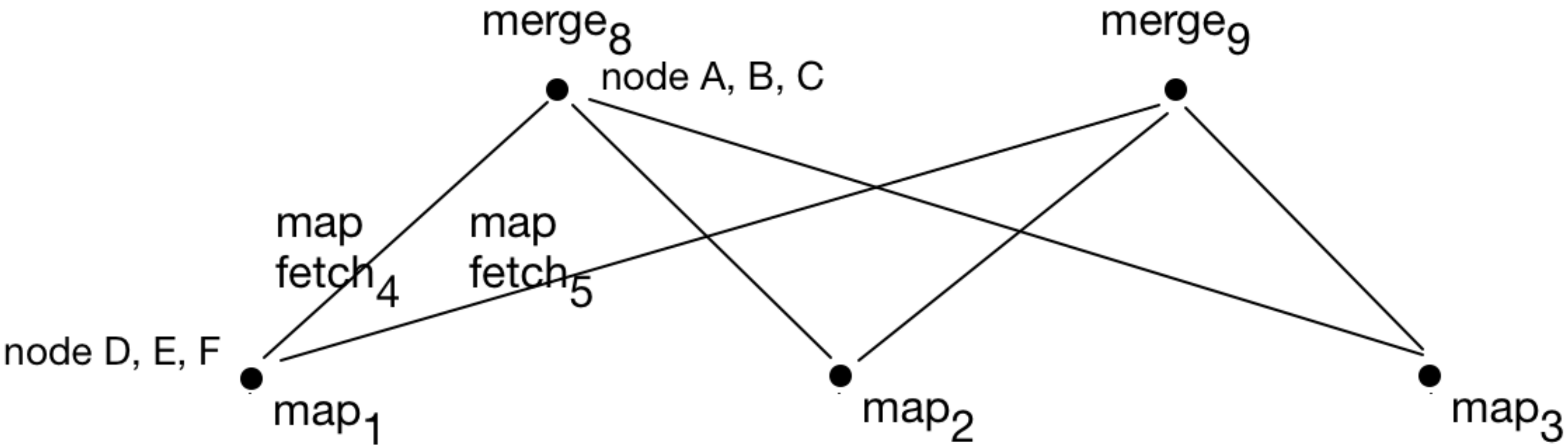
ExtendedOp (count(distinct cust_id))

Repartition (orders.cust_id)

Table (orders)

How to express
Repartition in
physical plan?

Repartition in Physical Plan



Takeaway

Logical Plan ≠ Physical Plan. A physical plan expresses your execution primitives. The way you define your distributed execution primitives impacts how coupled you are with “local execution”.

Different Executors for Different Workloads

Different Workloads

1. Simple Insert / Update / Delete / Select commands
   - High throughput and low latency
2. Real-time Select queries that get parallelized to hundreds of shards (<300ms)
3. Long running Select queries that join large tables
   - You can't restart a Select query just because one task (or one machine) in 1M tasks failed

Different Executors

1. Router Executor: Simple Insert / Update / Delete / Select commands
2. Real-time Executor: Real-time Select queries that touch 100s of shards (<300ms)
3. Task-tracker Executor: Longer running queries that need to scale out to 10K-1M tasks

Conclusions

- Distributed databases are about network I/O (and failure semantics).
- The Multi-Relational Algebra paper offers a complete theoretical framework to minimize network I/O.
- Citus maps that logical plan into a physical one that decouples local and distributed execution.
- Citus 5.1 is open source!

Questions

https://citusdata.com

https://github.com/citusdata/citus