Banking Case study: Scaling with low latency using NewSQL

Jags Ramnarayan (VMWare)
Jim Bedenbaugh (VMWare)

Qcon 2012

Agenda

- Business Requirements
  - Operational data
  - Analysis
- Problem Statement
  - Scaling pain
- Introduction to SQLfire
- Driving principles in SQLFire
- Use cases
- Demo (partitioned regions, colocation, etc)
- Data-aware procedures concepts
- Consistency model
- Shared-nothing disk persistence

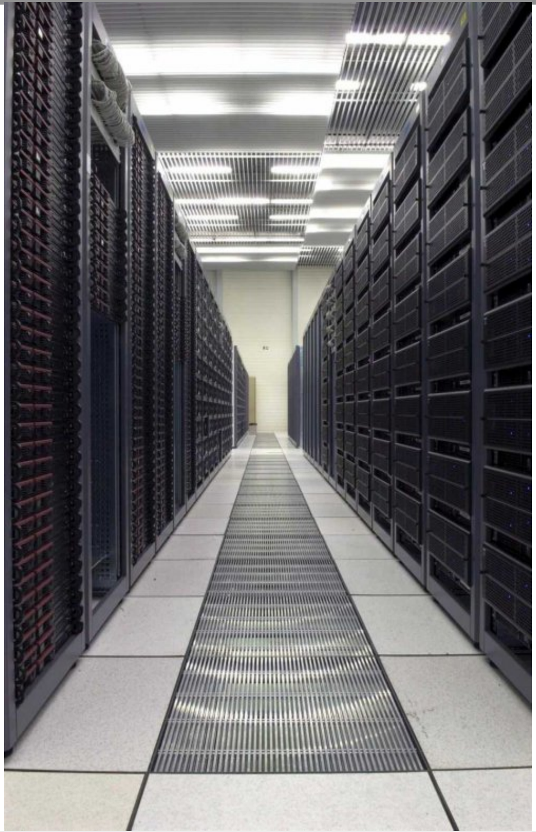
Business Requirements: What are they after?

- A large regional bank in the Northeastern U.S.
- Collects large amounts of operational data
  - By region and branch
  - Significant number of attributes associated with each transaction
    - Drive thru or foot traffic
    - Transaction type
    - Product types
    - Time of day

Business Requirements: What are they after? (con't)

- Analysis
  - The data is analyzed to determine the staffing requirements for each branch and region. These requirements yield guidance on:
    - Number of staff needed
    - Skills needed
    - Hours of operation

So what seems to be the problem?



"Right now our database is only 32G but...

"We are constantly acquiring new banks…

"And this database is growing rapidly…"

"And it takes at least a week to get new resources and this is just too slow."

Vertical Scaling has led to The “Jenga Architecture”

“We can only scale one way: Vertically. We want to scale horizontally but the vendor wants a pile of money to put in a new solution. When we need more resources, all we can do is jam in more memory and hang more drives off the same machine, but we’re getting at the end of that road. We need another solution.”

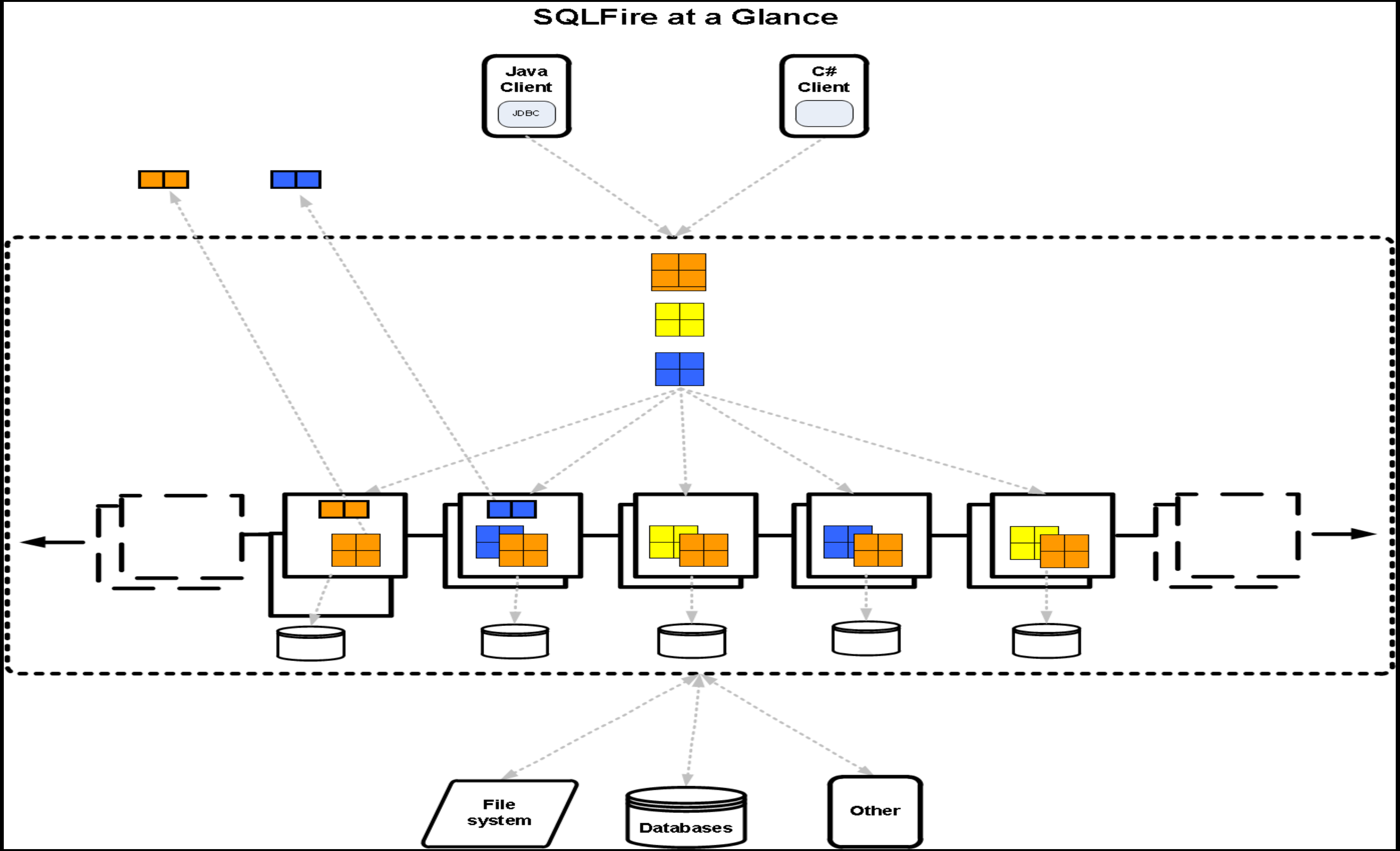
How did they arrive in this predicament?

Poor planning: “We just didn’t think about how this data is used or how much we would end up collecting over time.”

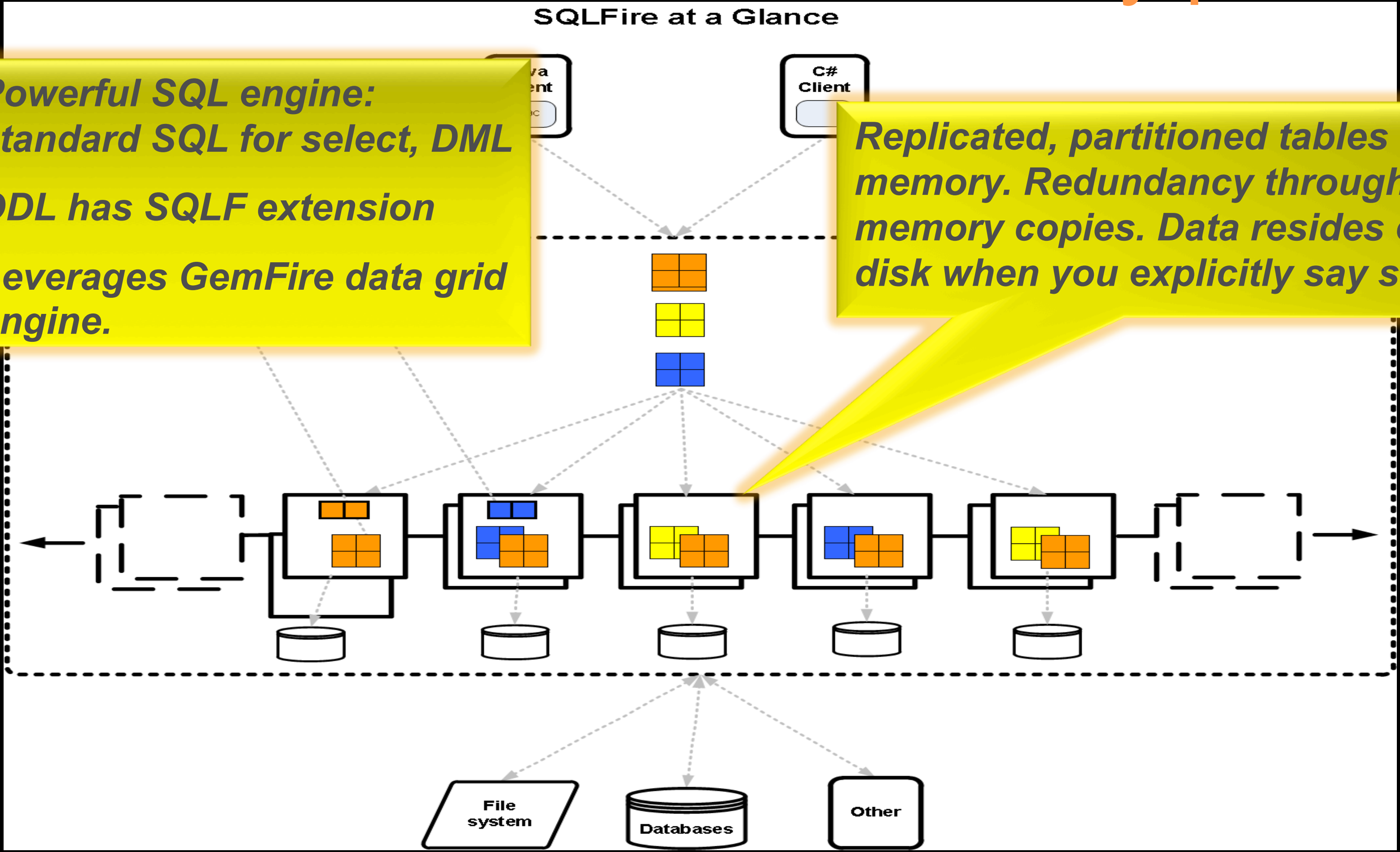
Doing it on the cheap: “We were locked into one database vendor and the original implementation was cheap to do with their low end database.”

Clustering: “Our team isn’t really all that sophisticated in doing these kinds of databases. An awful lot of our data lives on the mainframe.”

The introduction of SQLfire…



And how does SQLfire ease my pain?



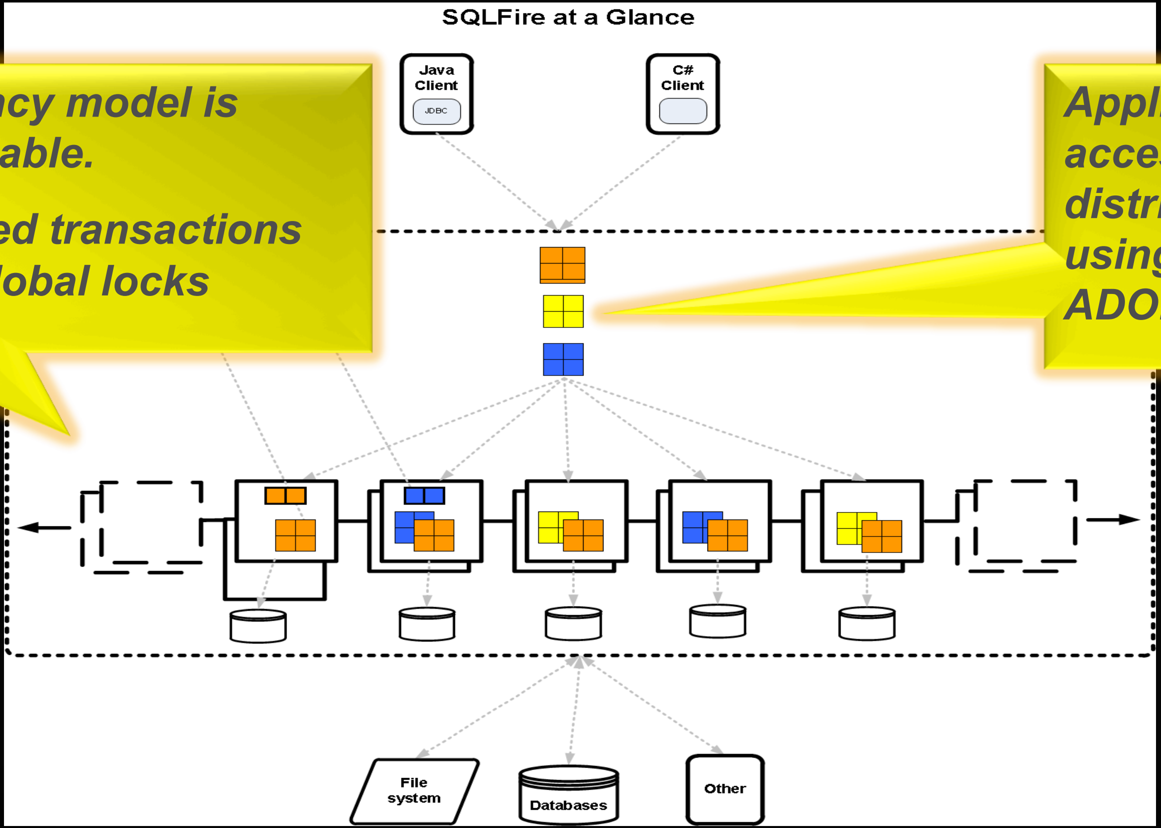
Powerful SQL engine: standard SQL for select, DML

DDL has SQLF extension

Leverages GemFire data grid engine.

Replicated, partitioned tables in memory. Redundancy through memory copies. Data resides on disk when you explicitly say so

Scaling at the speed of thought

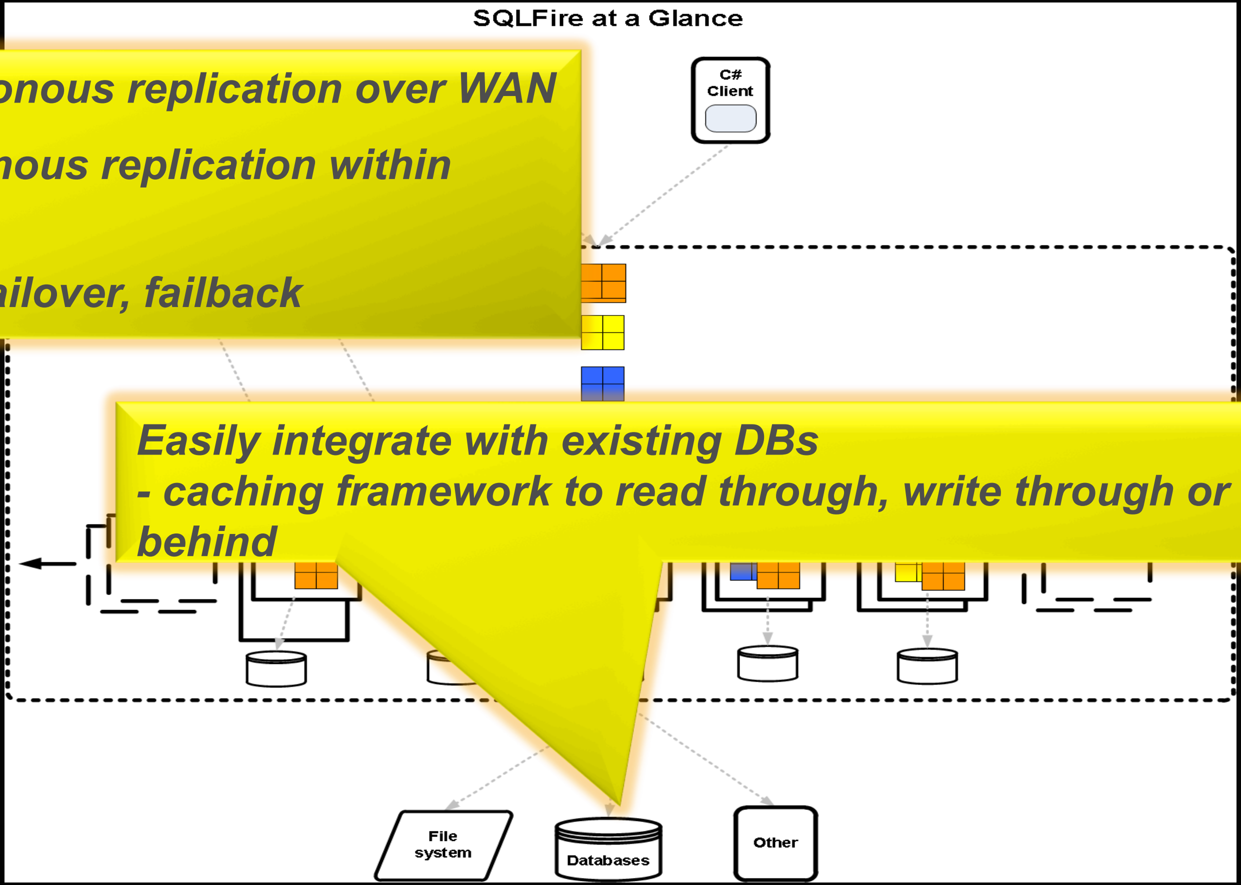


Consistency model is FIFO, Tunable.

Distributed transactions without global locks

Applications access the distributed DB using JDBC, ADO.NET

Scaling at the speed of thought



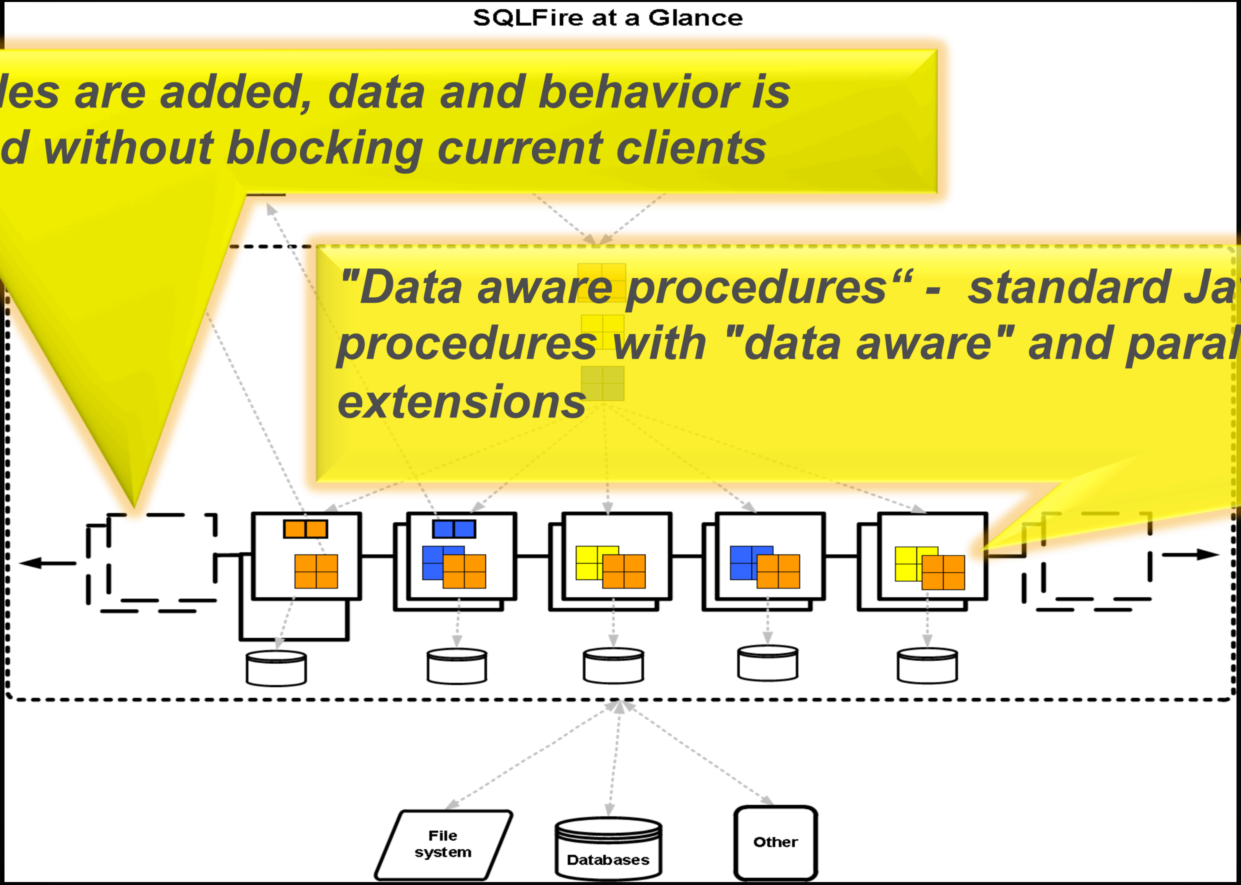
Asynchronous replication over WAN

Synchronous replication within cluster

Clients failover, failback

Easily integrate with existing DBs
- caching framework to read through, write through or write behind

Scaling at the speed of thought



When nodes are added, data and behavior is rebalanced without blocking current clients

"Data aware procedures" - standard Java stored procedures with "data aware" and parallelism extensions

The Partitioning Strategy: How we chose…

```
CREATE TABLE FLIGHTS
   (
     FLIGHT_ID CHAR(6) NOT NULL ,
     REGION INTEGER NOT NULL,
     SEGMENT_NUMBER INTEGER NOT NULL ,
     ORIG_AIRPORT CHAR(3),
     DEPART_TIME TIME, … )

PARTITION BY COLUMN(REGION)

REDUNDANCY 1

PERSISTENT;
```

Partitioning: The Result

What it looked like:

- 2x48G VM with 2 processors
- Data Partitioned and Replicated
- Split: 13 million rows/ 9 million rows

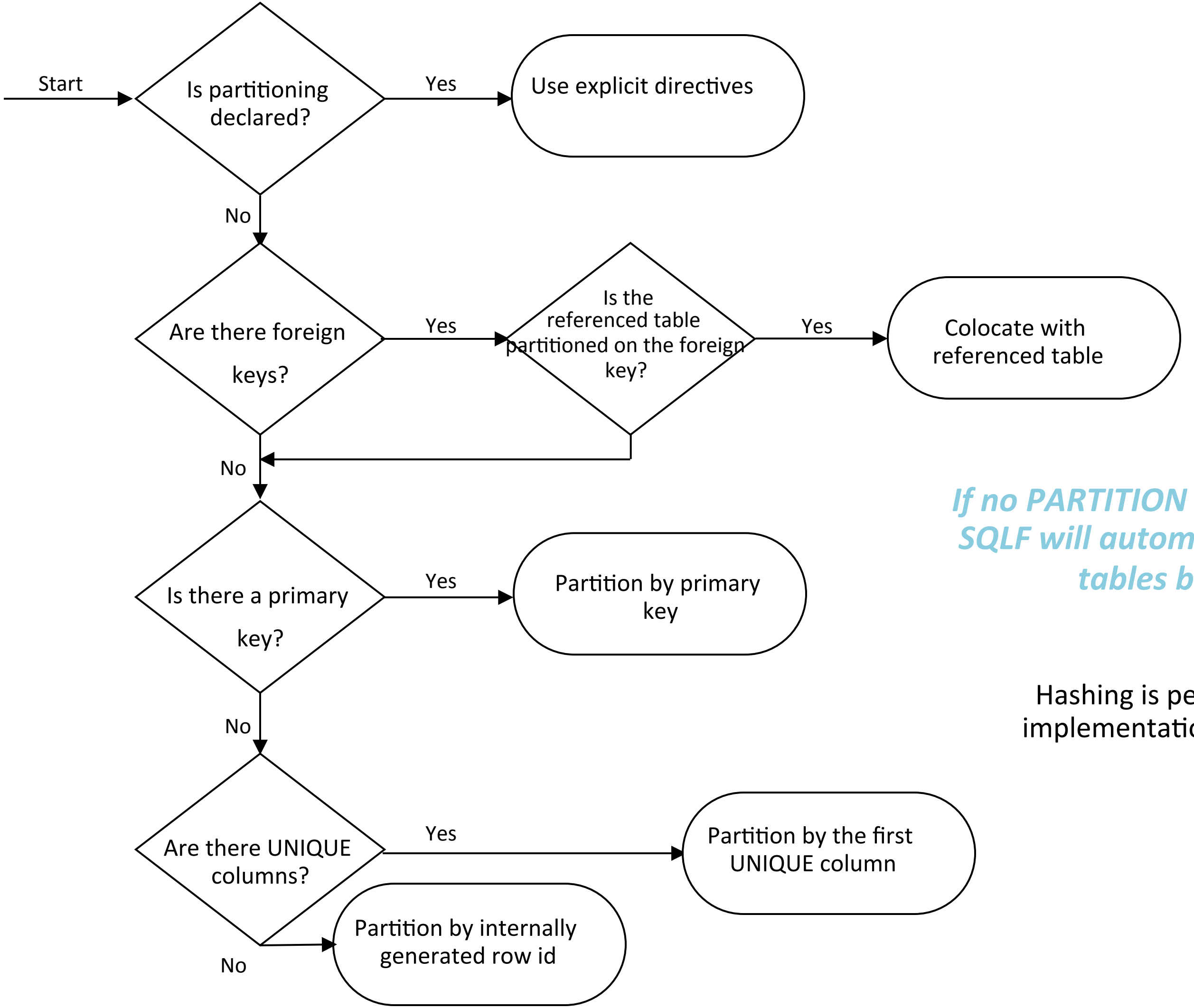
What happened when we added another VM

- Added 48g 2 processor
- Data rebalanced across 3 partitions: 8 million/6 million/8 million

How it performed

- We ran side by side comparisons of and existing SQL statement.
- The existing server took nearly 20 minutes to complete
- The SQLfire version completed in under 1 minute.

The benefit of partitioning is that we can go to a single partition and retrieve data instead of a table scan.



Start

Is partitioning declared?

Yes

Use explicit directives

No

Are there foreign keys?

Yes

Is the referenced table partitioned on the foreign key?

Yes

Colocate with referenced table

No

Is there a primary key?

Yes

Partition by primary key

No

Are there UNIQUE columns?

Yes

Partition by the first UNIQUE column

No

Partition by internally generated row id

If no PARTITION BY clause is specified, GemFire SQLF will automatically partition and collocate tables based on this algorithm.

Hashing is performed on the Java implementation of the column's type.

Reactions to the implementation

The DBA’s had the Grumpy Old Man response:

“Hey you kids get off my grass!”

Reactions to the implementation

Management response:

“Where do we sign?”

Reactions to the implementation

Business response:

“Where do we sign?”

Reactions to the implementation

Developers response:

“What? We have to modify existing SQL? This just doesn’t drop in?”

Conversion Gotchas and Tips…

DDL. It's different for Derby. DB2 has all kinds of options and parameters. Use a hatchet, not scissors when editing. I wrote a few scripts to rip out a lot of the DB2 DDL. It's just not needed.

Data types: Map them before you convert the DDL. Write a script to convert them.

Data conversion: SQLfire has a neat import procedure SYSCS_UTIL.IMPORT_TABLE. Use it. I always requested CSV files and split them up into chunks in case anything went wrong.

Use JDBCRowloader for read misses. Comes with SQLfire.
Use DDLUtils for DDL conversion.
Cannot use Stored Procedures. Rewrite as Java Stored Procedure

SQLFire Driving Principles

NoSQL data models less rigid but most now support some form of SQL – cql, un-ql, oql, etc♪
SQL : Flexible, easily understood, strong type system ♪
essential for query engine efficiency♪

Focus on commodity servers; ♪
Memory density follows Moore's law♪
Optimize for memory; Focus on large Not "Big data"

Undifferentiated features in next gen databases – ♪
Horizontal scalability, high availability

SQLFire Driving Principles

Data is flowing.. Work with relevant, “NOW” data

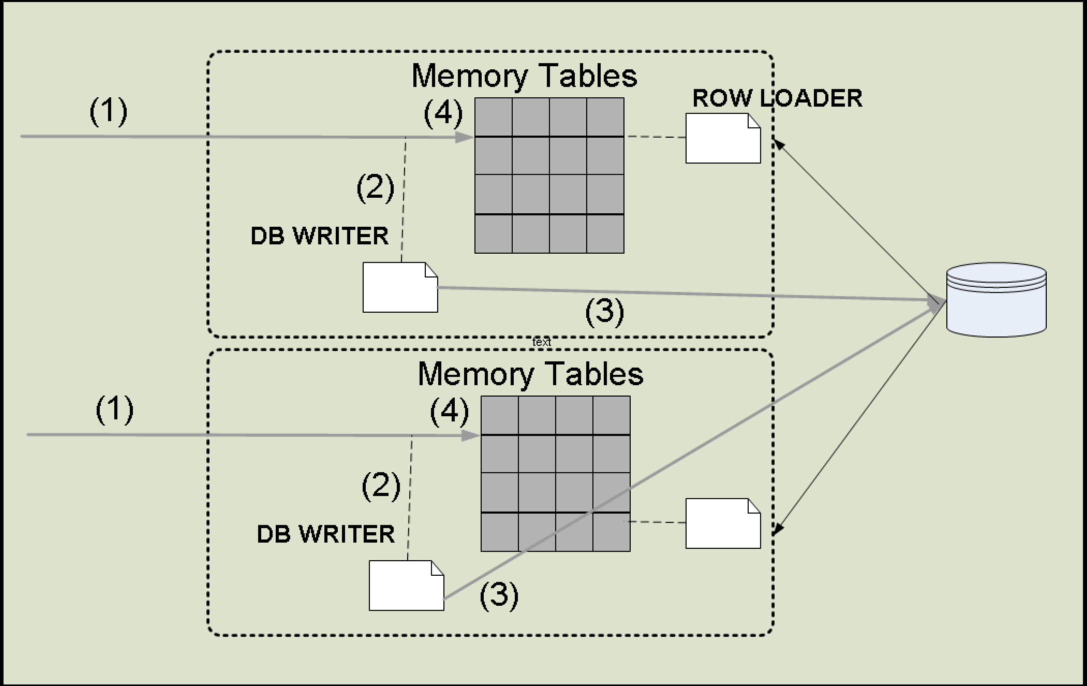
Not Just High Availability.. Continuous availability♪
Synchronous copies in proximity.. Async copies across WAN

Exploit data affinity for parallel processing; offer new APIs♪
– App developer is the new DBA♪

Consistency should be tunable♪
Eventual consistency is too difficult for the average developer♪
Write(A,2) → Read(A) may return 1 or (1,2)

DESIGN PATTERNS

“Write thru” Distributed caching



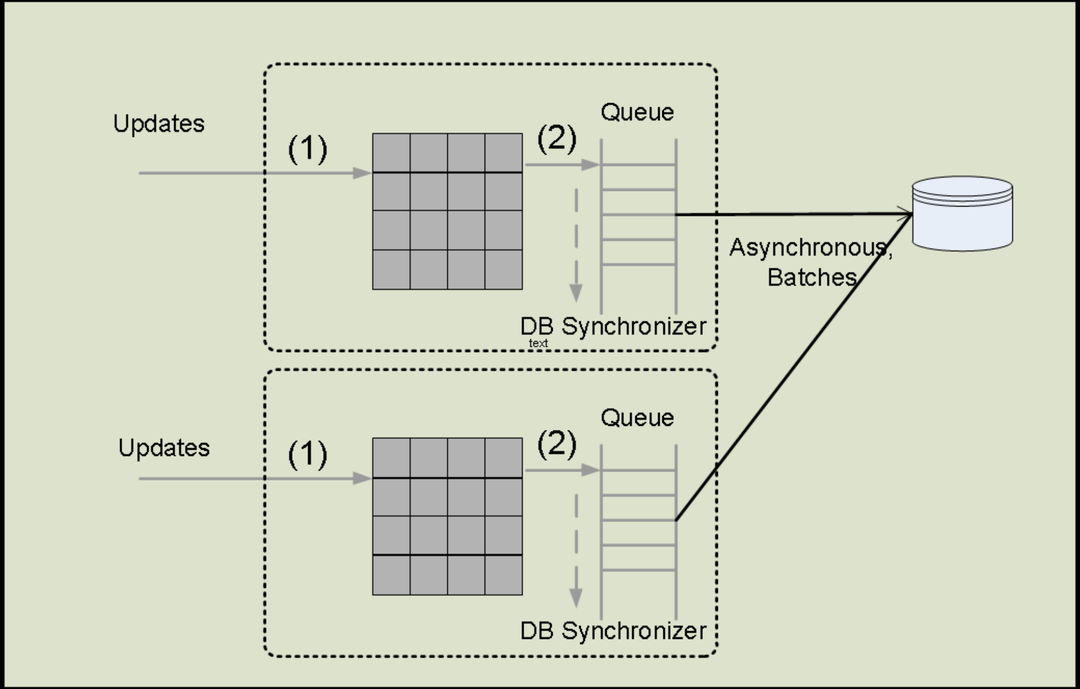
Pre-load using DDLUtils for queries

Lazily load using “RowLoader” for PK queries

Configure LRU eviction or expiry for large data

“Write thru” – participate in container transaction

Distributed caching with Async writes to DB



Buffer high write rate from DB

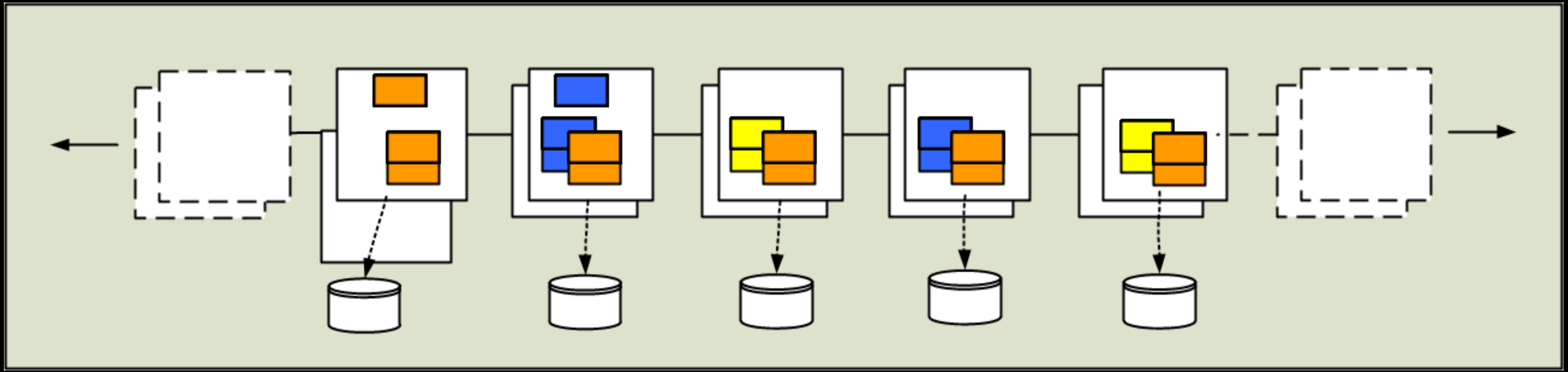
Writes can be enqueued in memory redundantly on multiple nodes

Or, also be persisted to disk on each node

Batches can be conflated and written to DB

Pattern for “high ingest” into Data Warehouse

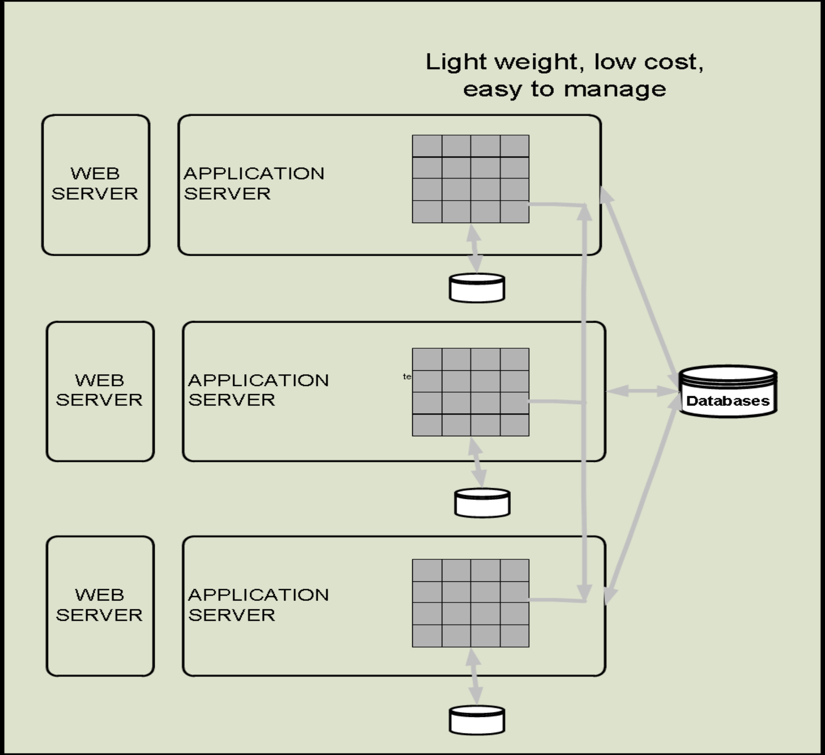
As a scalable OLTP data store



Shared nothing persistence to disk
Backup and recovery

No Database to configure and be throttled by

As embedded, clustered Java database



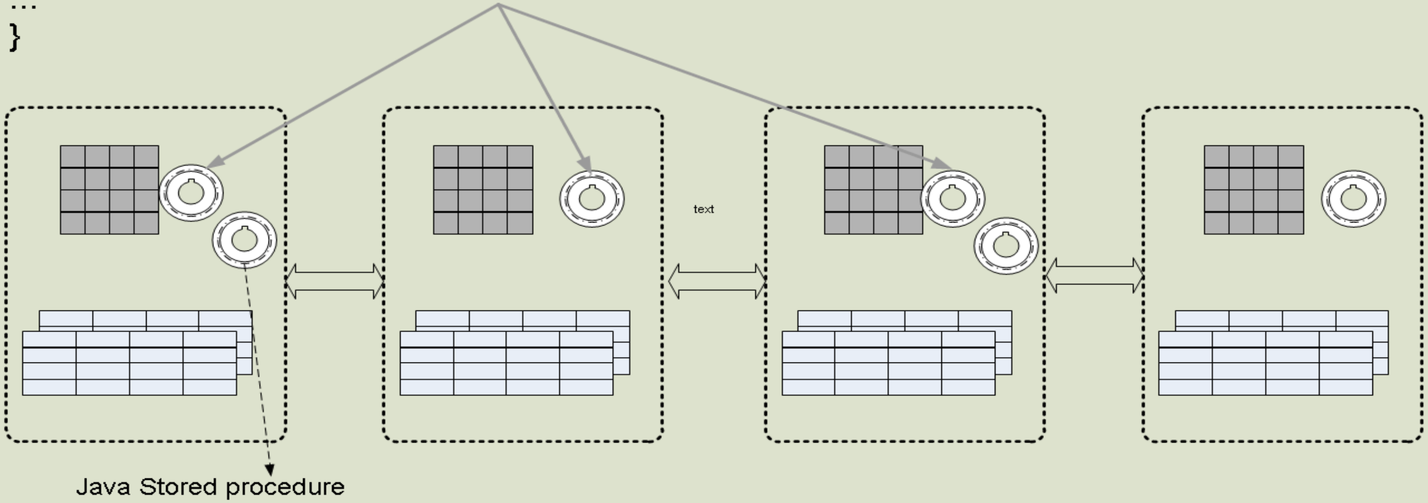
Just deploy a JAR or WAR into clustered App nodes

Just like H2 or Derby except data can be sync'd with DB is partitioned or replicated across the cluster

Low cost and easy to manage

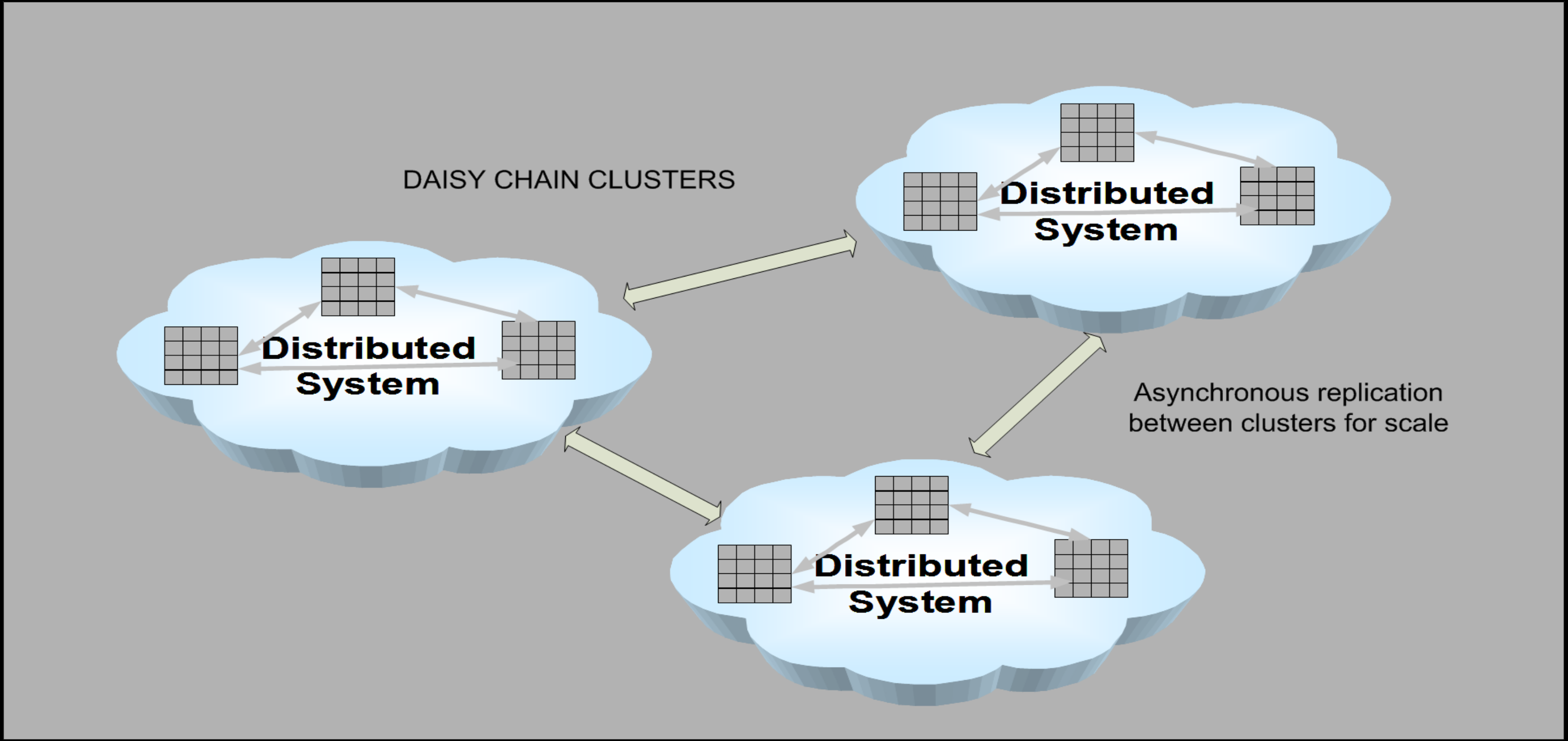
To process app behavior in parallel

```
@ParititionedTable(TableName="trades")
public List AnalyzeTrades(@FilterKey Set<String> months, String portfolio) { …
…
}
```



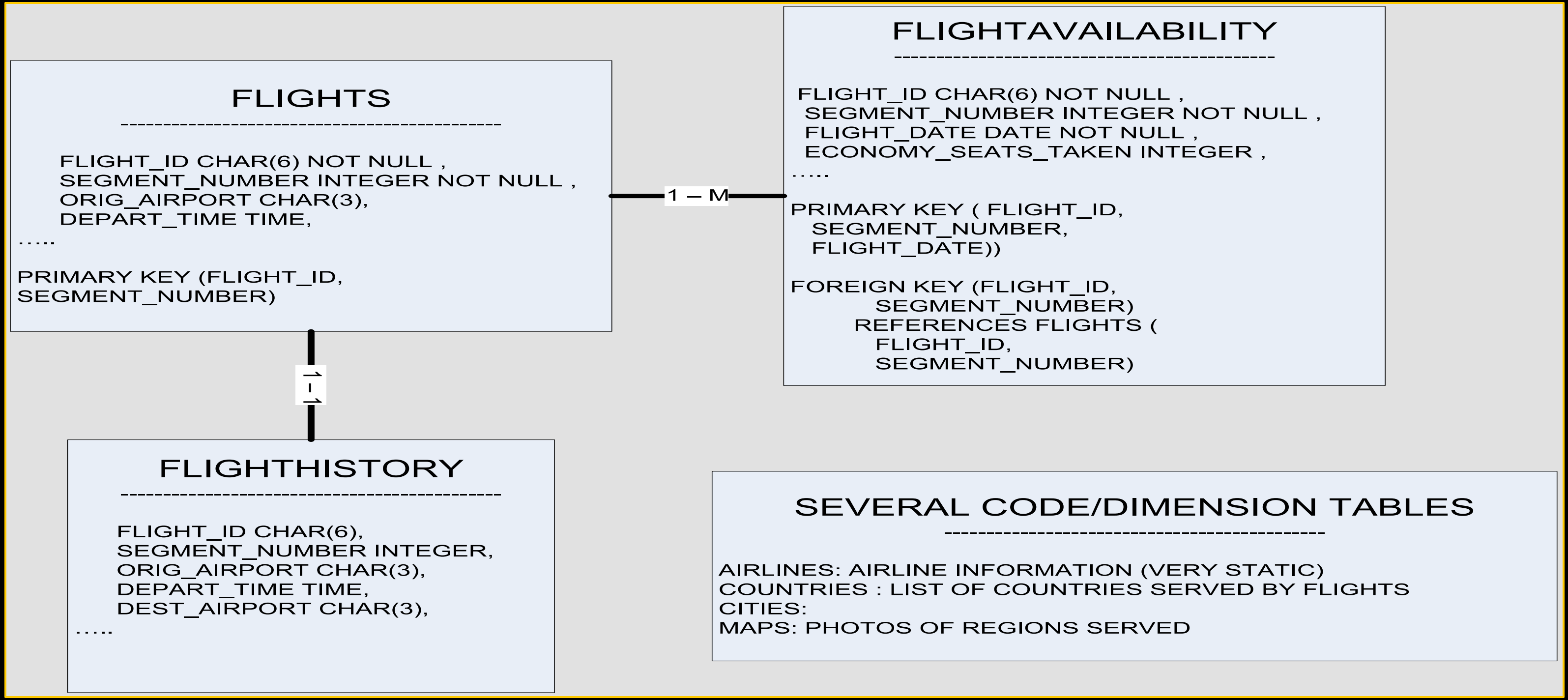
Map-reduce but based on simpler RPC

To make data visible across sites in real time



Demo

default partitioned tables, colocation, persistent tables



FLIGHTS

```
FLIGHT_ID CHAR(6) NOT NULL ,
SEGMENT_NUMBER INTEGER NOT NULL ,
ORIG_AIRPORT CHAR(3),
DEPART_TIME TIME,
…..

PRIMARY KEY (FLIGHT_ID,
SEGMENT_NUMBER)
```

FLIGHTAVAILABILITY

```
FLIGHT_ID CHAR(6) NOT NULL ,
 SEGMENT_NUMBER INTEGER NOT NULL ,
 FLIGHT_DATE DATE NOT NULL ,
 ECONOMY_SEATS_TAKEN INTEGER ,
…..

PRIMARY KEY ( FLIGHT_ID,
   SEGMENT_NUMBER,
   FLIGHT_DATE))

FOREIGN KEY (FLIGHT_ID,
            SEGMENT_NUMBER)
         REFERENCES FLIGHTS (
            FLIGHT_ID,
            SEGMENT_NUMBER)
```

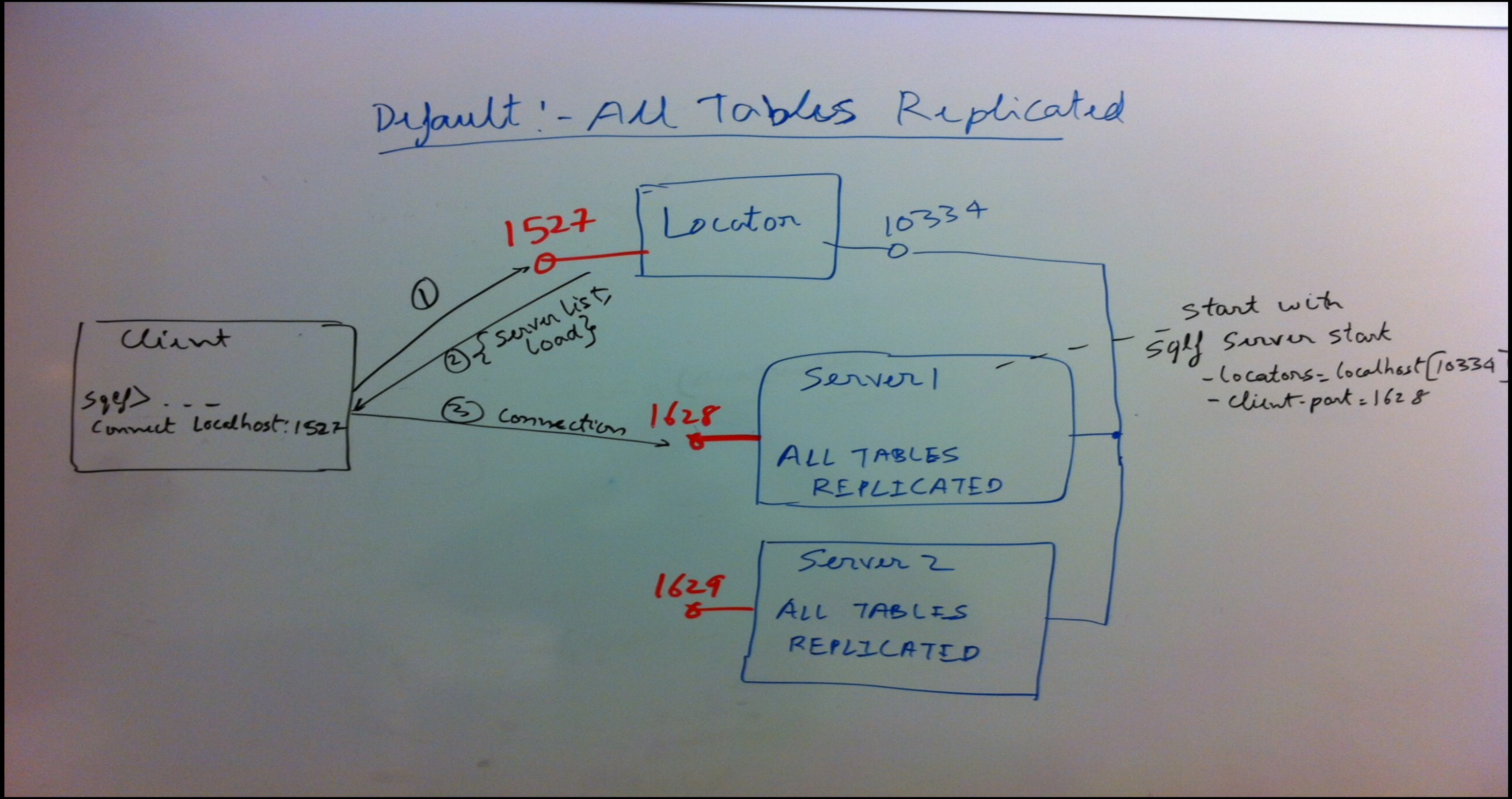
FLIGHTHISTORY

```
FLIGHT_ID CHAR(6),
SEGMENT_NUMBER INTEGER,
ORIG_AIRPORT CHAR(3),
DEPART_TIME TIME,
DEST_AIRPORT CHAR(3),
…..
```

SEVERAL CODE/DIMENSION TABLES

AIRLINES: AIRLINE INFORMATION (VERY STATIC)
COUNTRIES : LIST OF COUNTRIES SERVED BY FLIGHTS
CITIES:
MAPS: PHOTOS OF REGIONS SERVED

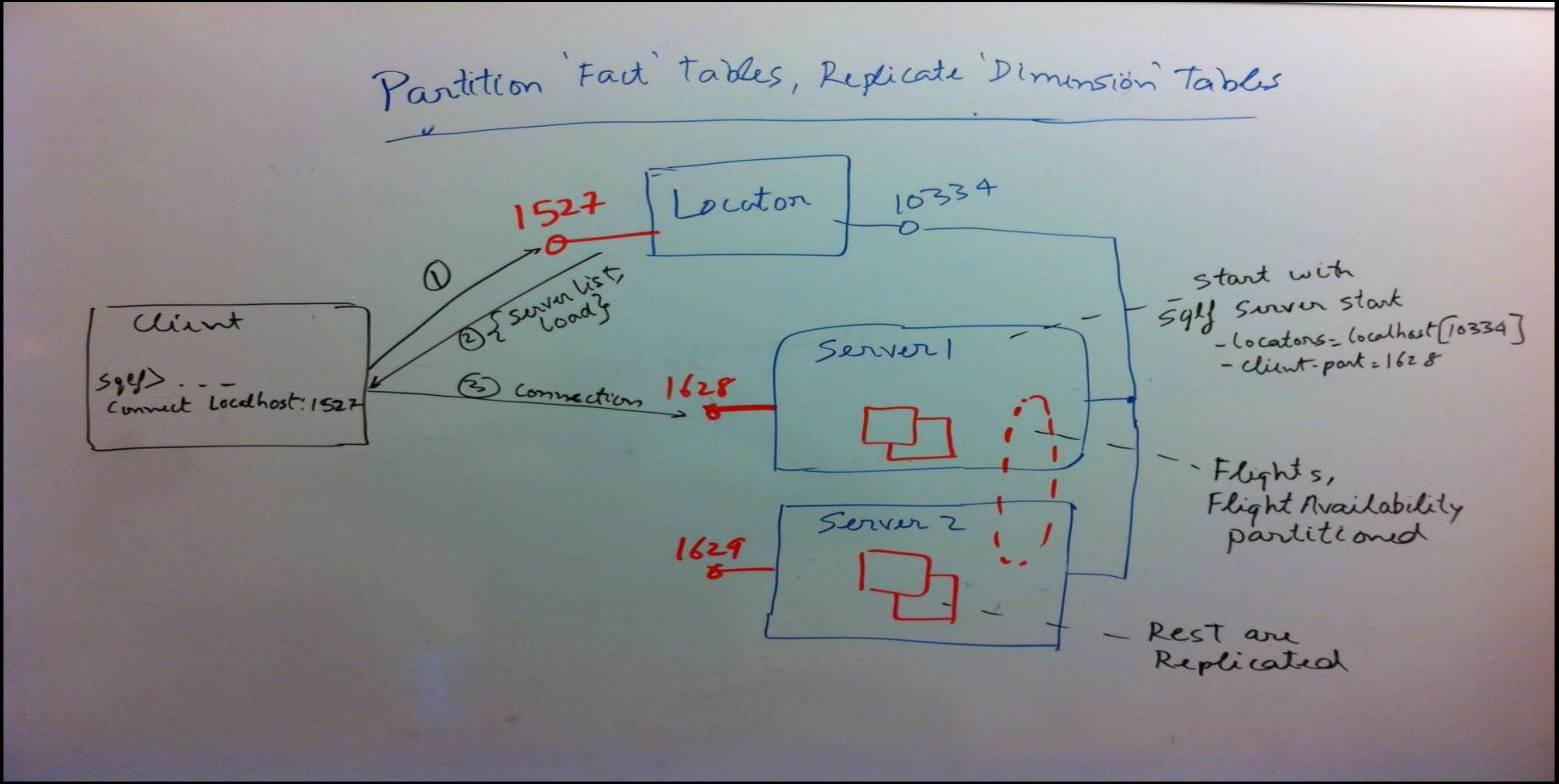
Demo – Start with replicated tables



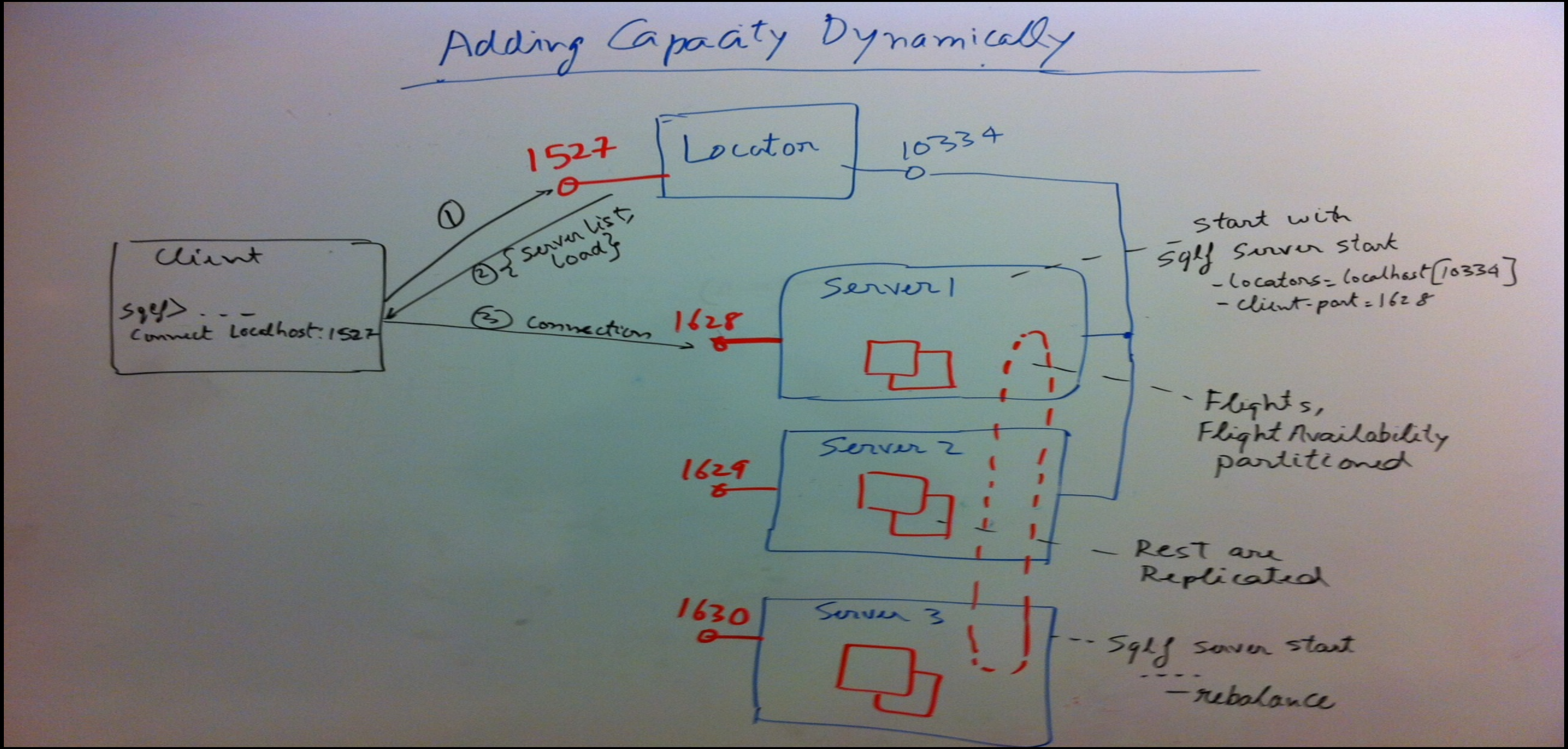
start with

```
sqlf server start
  -locators=localhost[10334]
  -client-port=1628
```

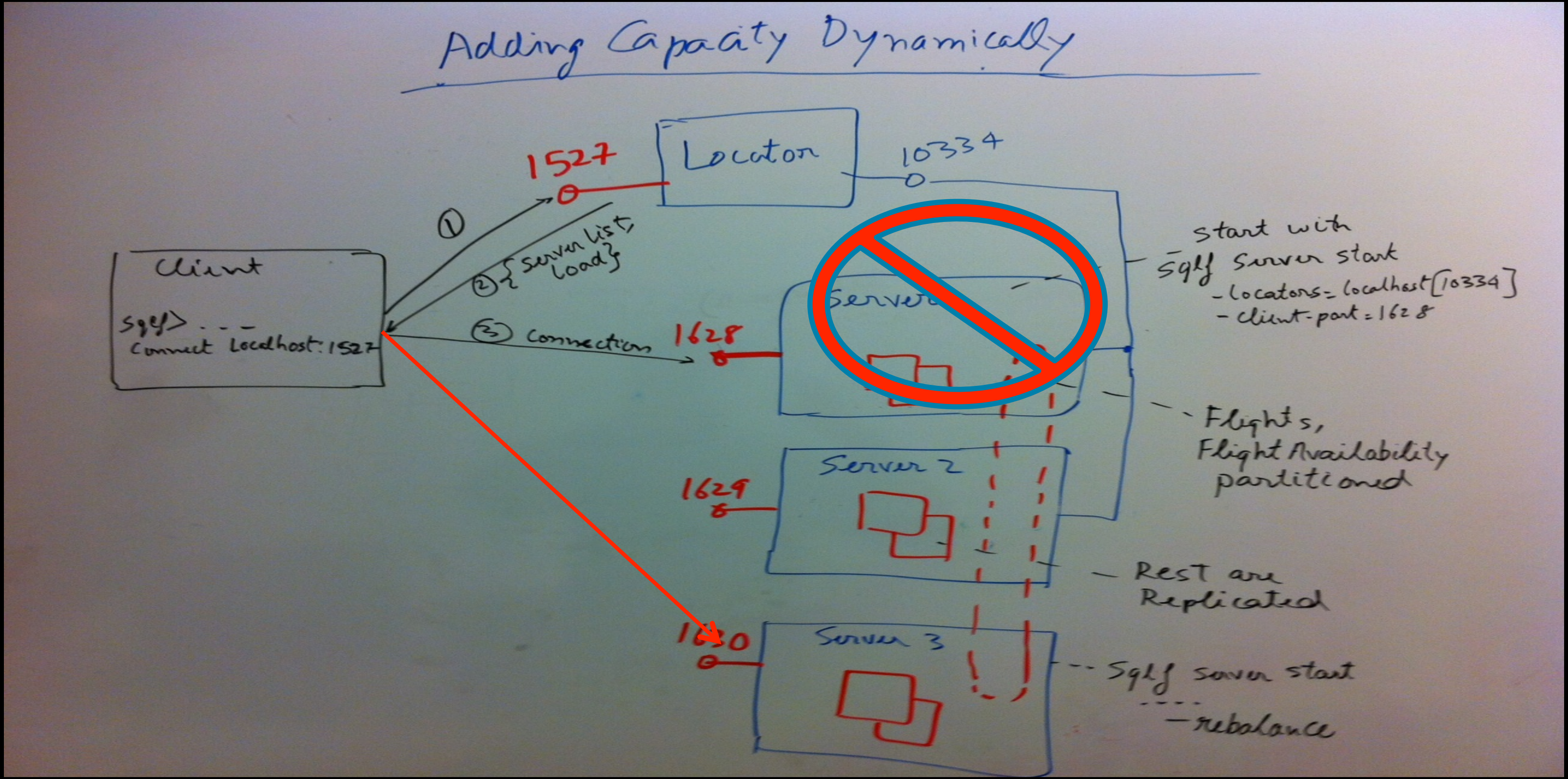
Demo – Partition the “fact” tables



Demo – Add a new server and rebalance



Demo – HA


Adding Capacity Dynamically
Locator
1527
10334
①
② {server list, load}
③ connection
Client
sqlf> . . .
Connect Localhost:1527
1628
Server
1629
Server 2
1630
Server 3
start with
sqlf server start
-locators=localhost[10334]
-client-port=1628
Flights,
Flight Availability
partitioned
Rest are
Replicated
Sqlf server start
. . . .
-rebalance

Linearly scaling joins

Say, Flights and FlightAvailability both were hash partitioned on PK

```
Select * from Flights f, FlightAvailability fa
where f.flight_id = fa.flight_id
and f.flight_id ='xxx‘ and fa.seats_taken > yy;
```

- With Hash partitioning the join would have to execute everywhere

- Distributed joins are expensive and inhibit scaling
  - joins across distributed nodes could involve distributed locks and potentially a lot of intermediate data transfer across nodes

Partition Aware DB Design

Designer thinks about how data maps to partitions

- The main idea is to:

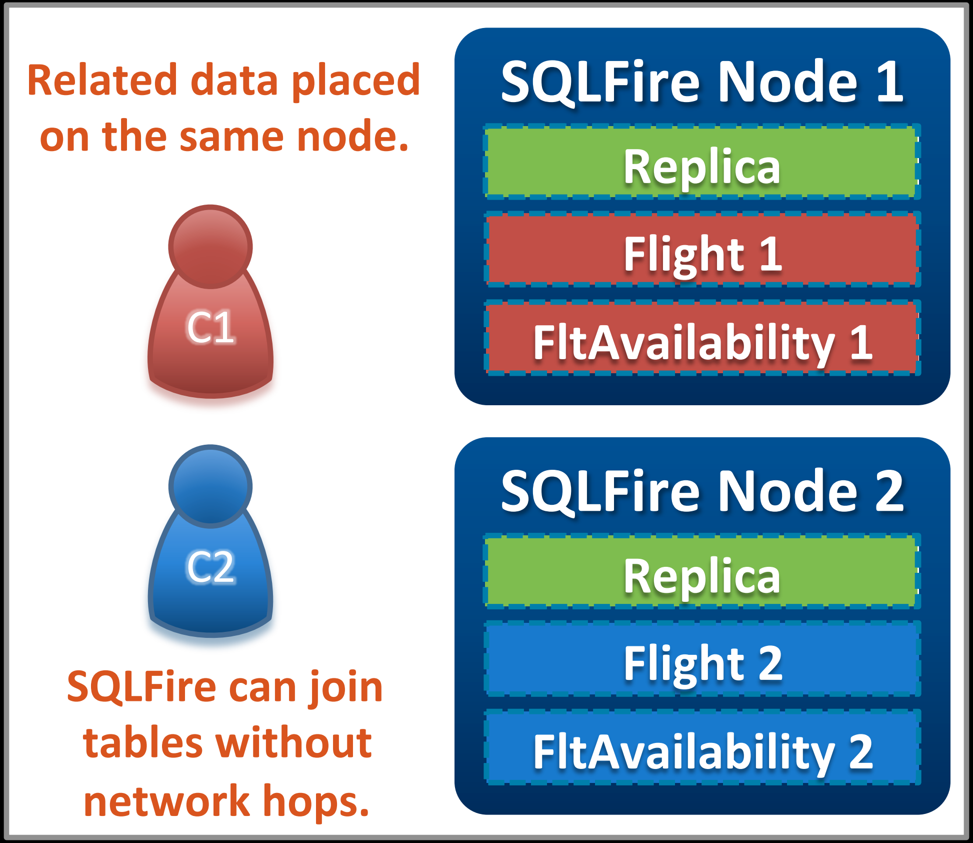
1) **minimize excessive data distribution by keeping the most frequently accessed and joined data collocated on partitions**

2) **Collocate transaction working set on partitions so complex 2-phase commits/paxos commit is eliminated or minimized.**

Read Pat Helland's "Life beyond Distributed Transactions" and the Google MegaStore paper

Collocate Data For Fast Joins.

```
CREATE TABLE FlightAvailability
    (flight_id ..,
    segment .., date ..)
PARTITION BY
    COLUMN (flight_id)
    COLOCATE WITH Flights;
```

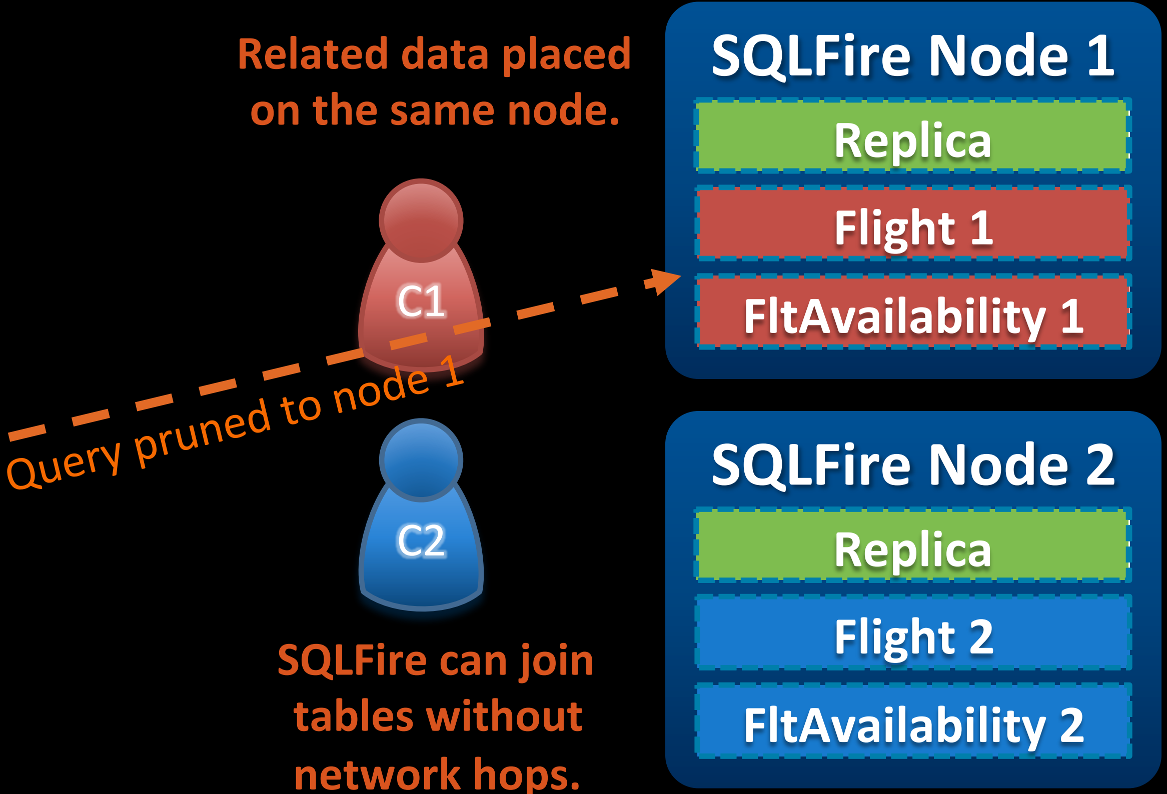


Related data placed on the same node.

SQLFire can join tables without network hops.

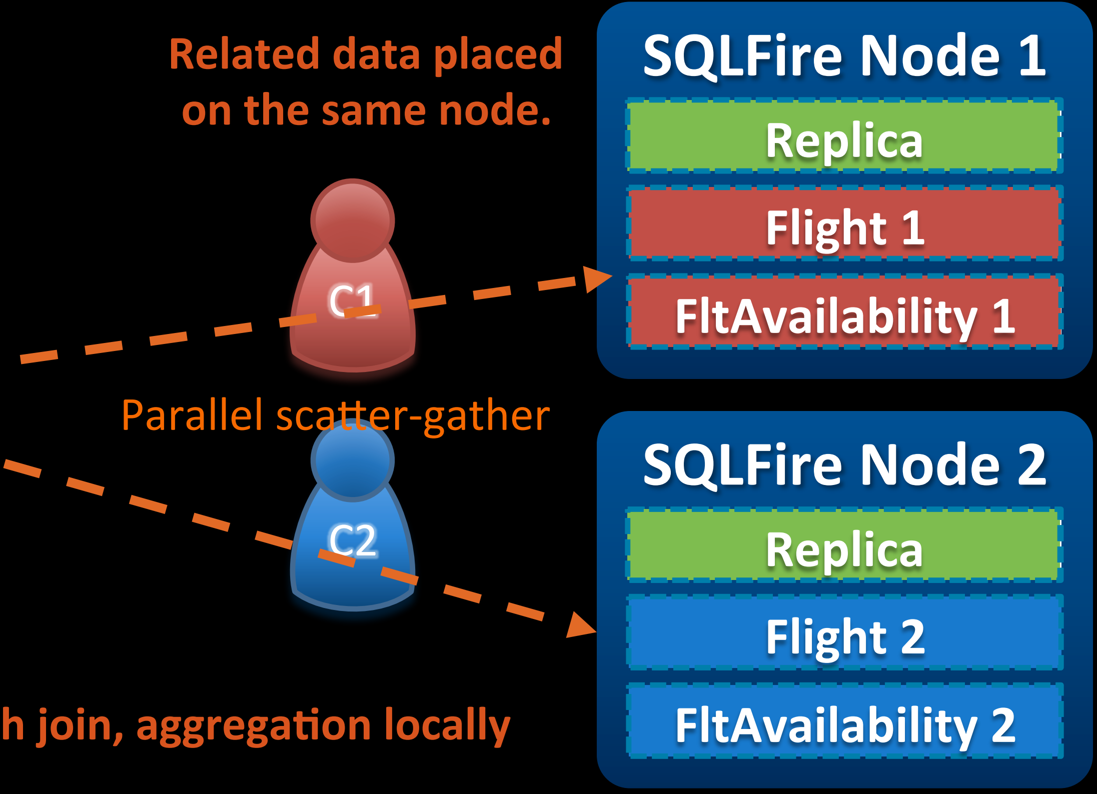
Collocate Data For Fast Joins.

*Select * from*
Flights f, FlightAvailability fa
where <equijoin clause>
and f.flight_id ='UA765';



Collocate Data For Fast Joins.

```
SELECT sum(fa.seats_taken),
f.orig_airport, fa.date
FROM flights f, FltAvailability fa
WHERE <equijoin>
GROUP By fa.date, f.orig_airport
ORDER BY fa.date, f.orig_airport
DESC
```

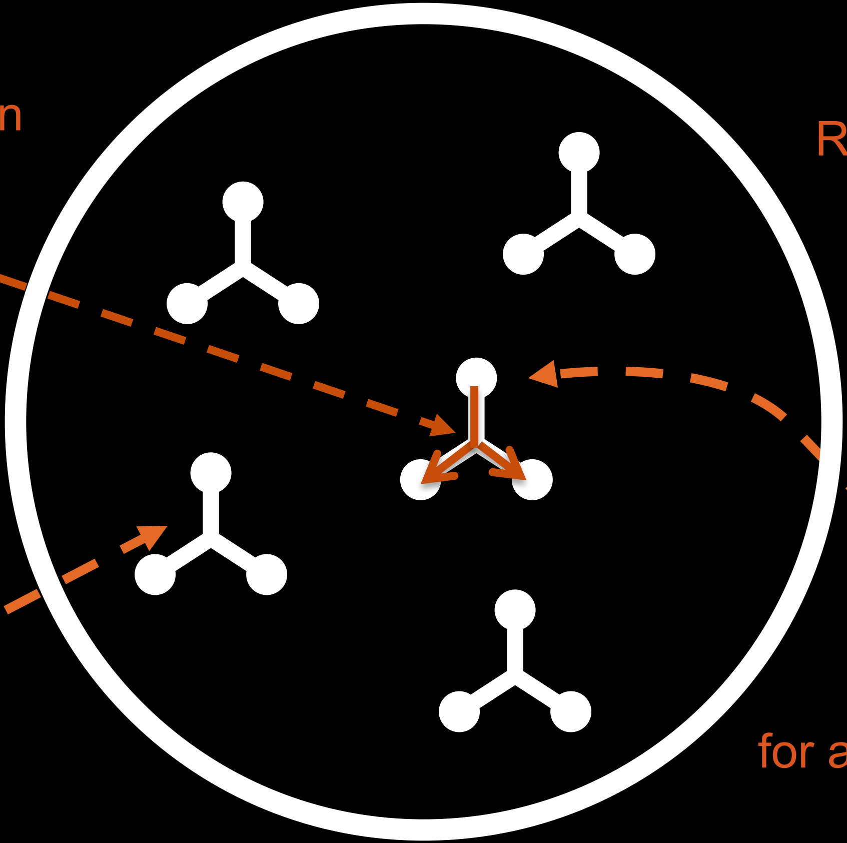


Related data placed on the same node.

Parallel scatter-gather

In parallel, each node does hash join, aggregation locally

Partitioning and redundancy



Replication is synchronous but done in parallel

Replication can be "rack aware"

Redundancy = 2 (but tunable)

Single owner for any row at point in time

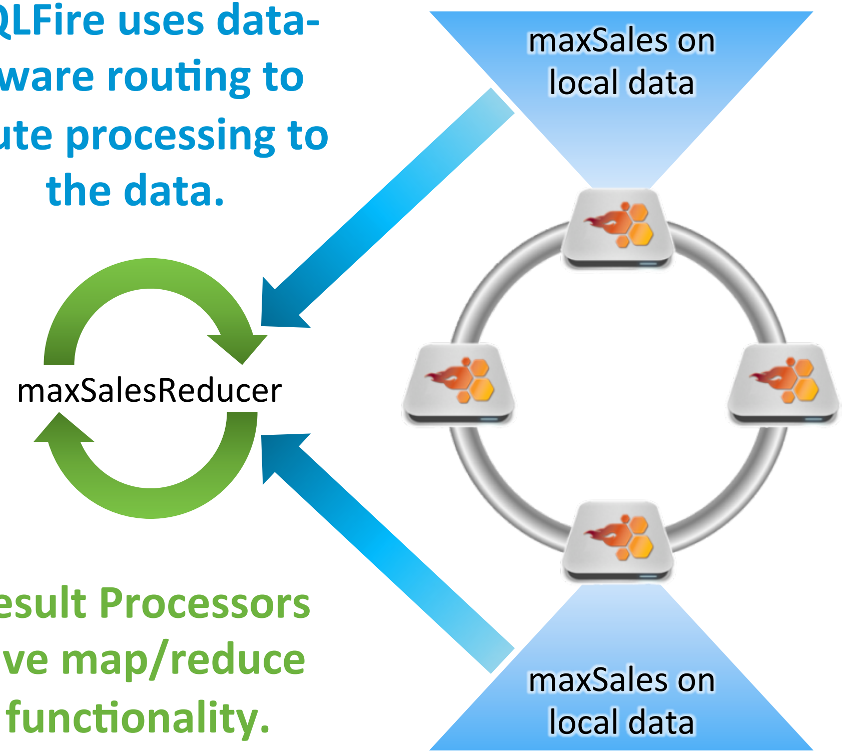
Data-Aware Stored Procs

- Procedure execution routed to the data
- Full scaled-out execution
- Highly available
- Use pure Java to access/store data
- Demo later on

Like Map/Reduce But Different

Scaling Stored Procedures

```
CALL maxSales(arguments)
 ON TABLE sales
WHERE (Location in (‘CA’,’WA’,’OR')
WITH RESULT PROCESSOR
maxSalesReducer
```



SQLFire uses data-aware routing to route processing to the data.

Result Processors give map/reduce functionality.

Scalability: Consistency

With Transactions

- Distributed transactions with 1-phase commit
  - Coordinator per node
  - Eager locking + Fail fast

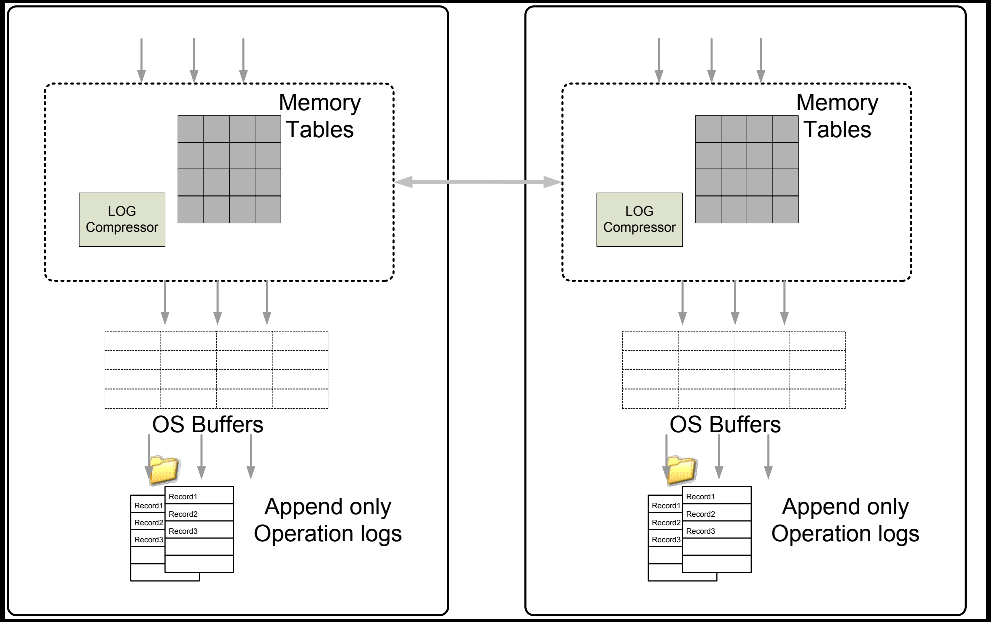
Assumes:
Most x-actions small in space and time
Write-write conflicts rare

And Without

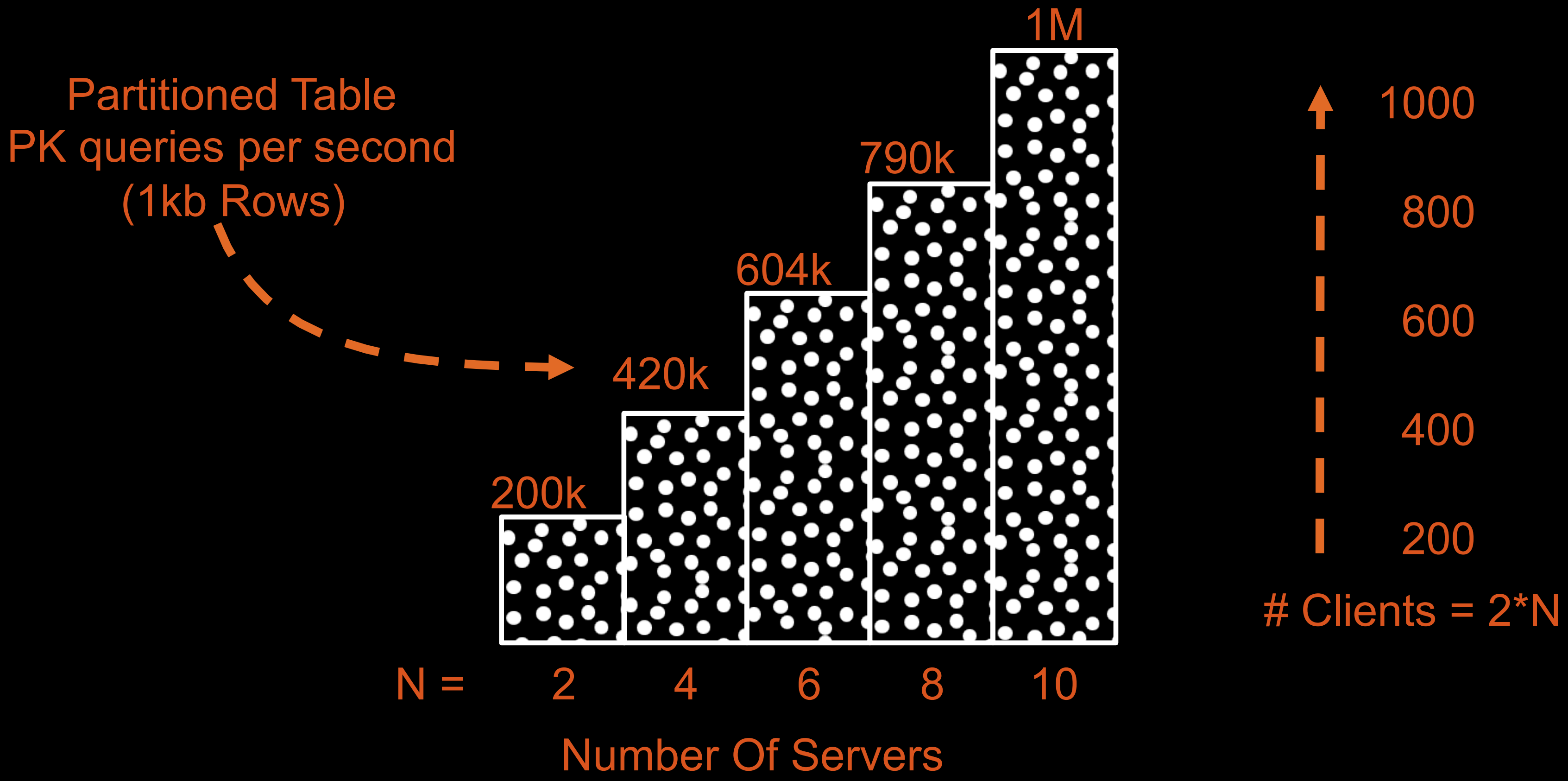
- Row updates always atomic and isolated
- FIFO consistency

Scalability: High performance persistence

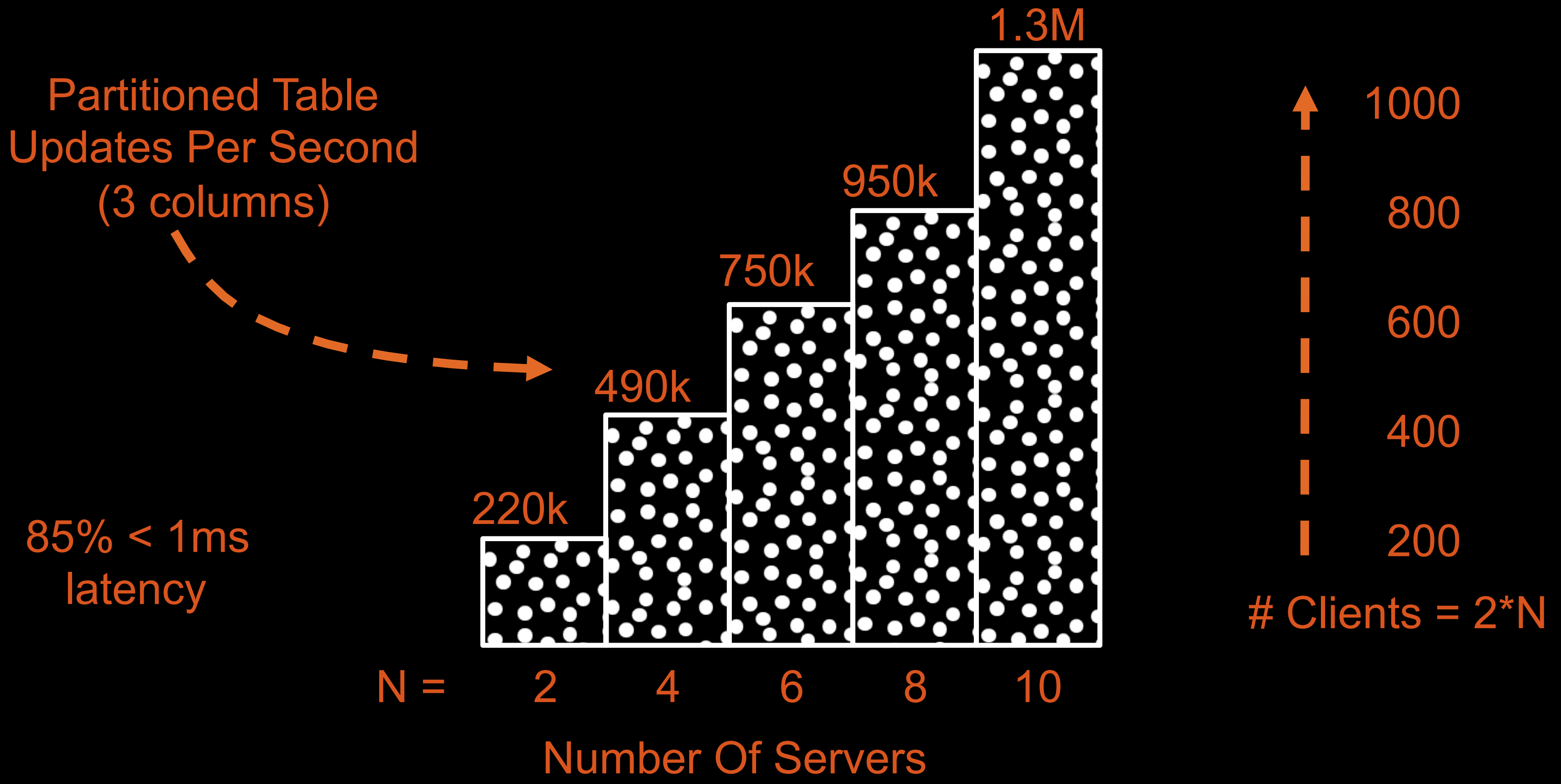
- Parallel log structured storage
- Each partition writes in parallel
- Backups write to disk also
  - Increase reliability against h/w loss



How does it scale for queries?



How does it scale for updates?



Q & A

Download: http://vmware.com/go/sqlfire

Try SQLFire Today!

Free for developer (3 nodes) perpetually.

:sigh:
Just Google it

Forum: http://vmware.com/vmtn/appplatform/vfabric_sqlfire

Got questions? Get answers.

Twitter: @vFabricSQLFire

vmware SQLFire