ODC

Beyond The Data Grid: Coherence, Normalisation, Joins and Linear Scalability

Ben Stopford : RBS

The Story…

The internet era has moved us away from traditional database architecture, now a quarter of a century old.

Industry and academia have responded with a variety of solutions that leverage distribution, use of a simpler contract and RAM storage.

We introduce ODC a NoSQL store with a unique mechanism for efficiently managing *normalised* data.

We show how we adapt the concept of a Snowflake Schema to aid the application of replication and partitioning and avoid problems with distributed joins.

Finally we introduce the 'Connected Replication' pattern as mechanism for making the star schema practical for in memory architectures.

The result is a highly scalable, in-memory data store that can support both millisecond queries and high bandwidth exports over a normalised object model.

Database Architecture is Old

Most modern databases still follow a 1970s architecture (for example IBM's System R)

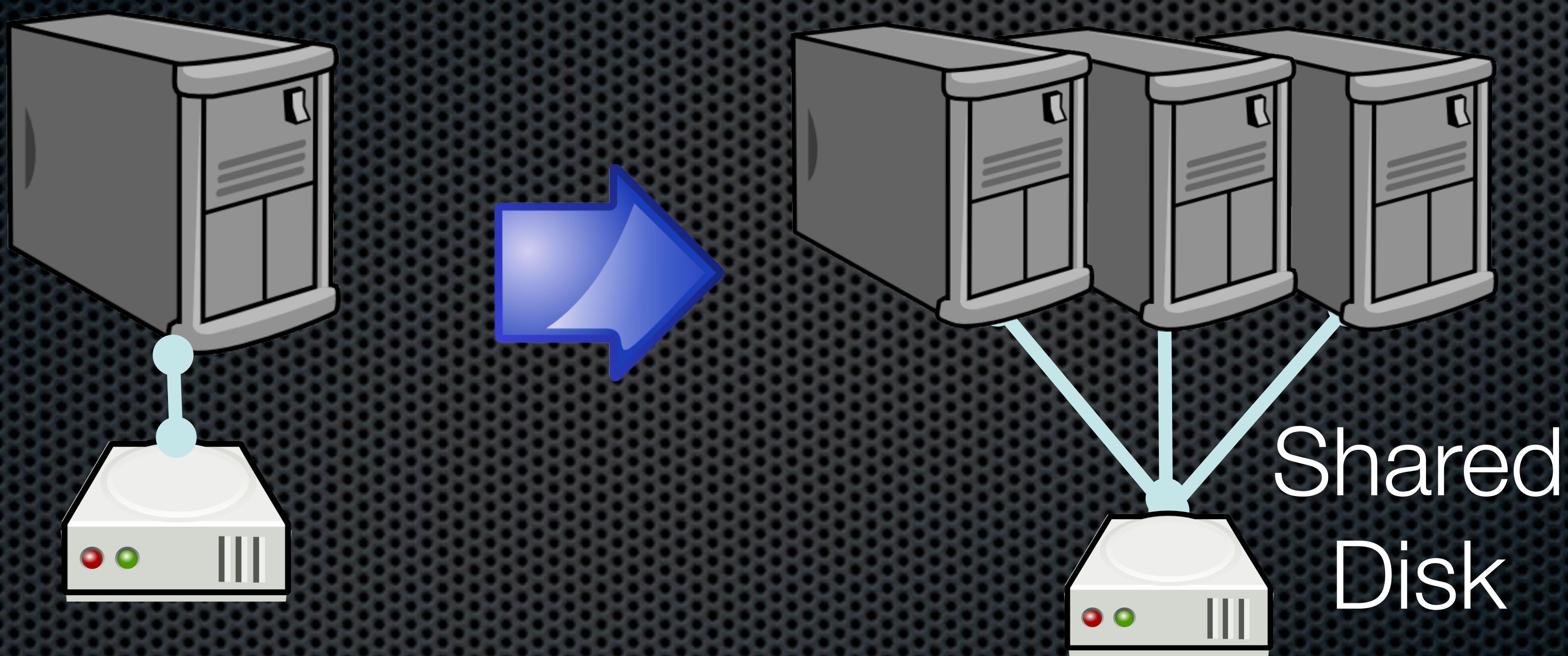
“Because RDBMSs can be beaten by more than an order of magnitude on the standard OLTP benchmark, then there is no market where they are competitive. As such, they should be considered as legacy technology more than a quarter of a century in age, for which a complete redesign and re-architecting is the appropriate next step.”

Michael Stonebraker (Creator of Ingres and Postgres)

What steps have we taken to improve the performance of this original architecture?

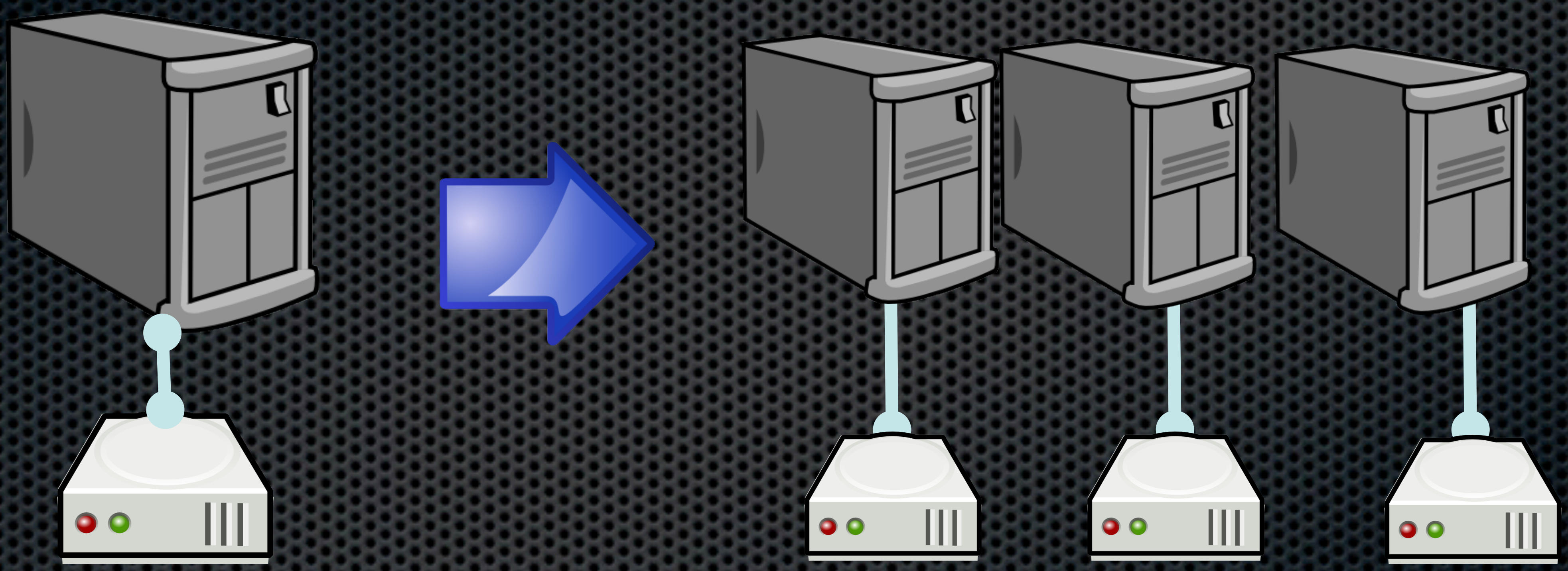
Improving Database Performance (1)

Shared Disk Architecture



Improving Database Performance (2)

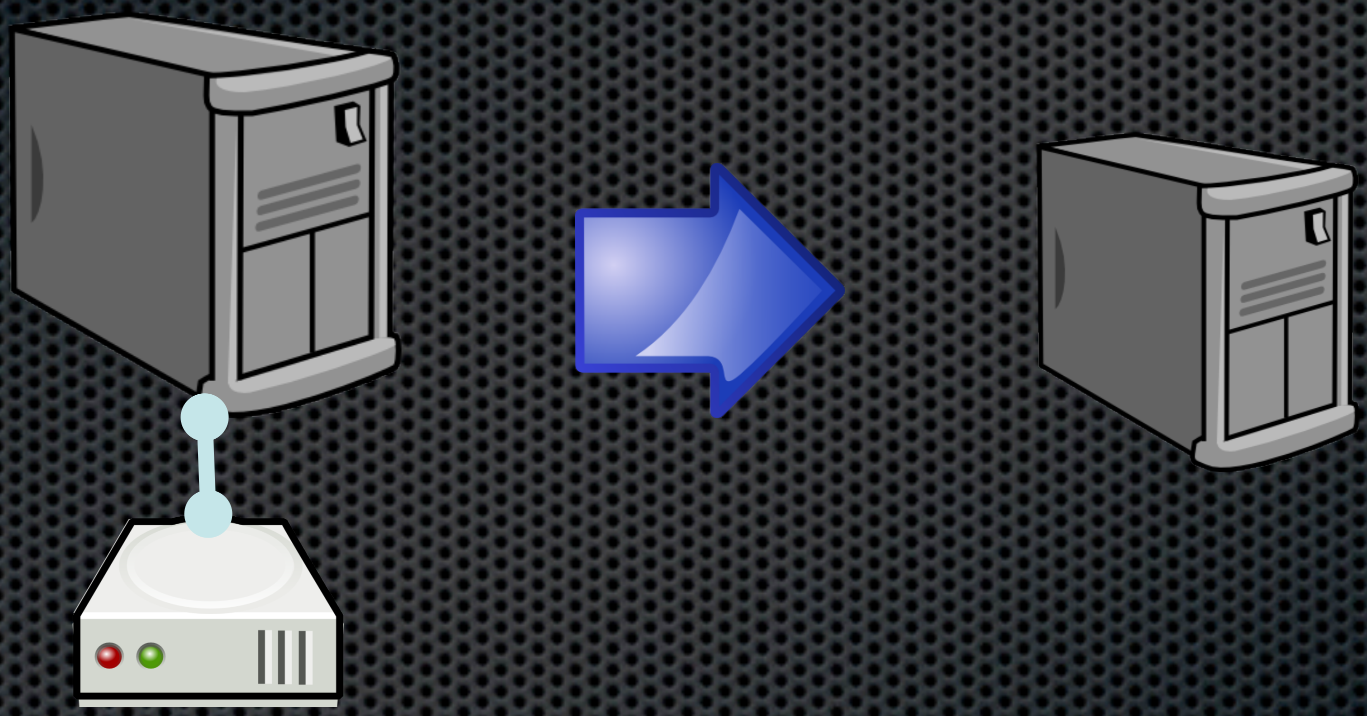
Shared Nothing Architecture



Shared Nothing

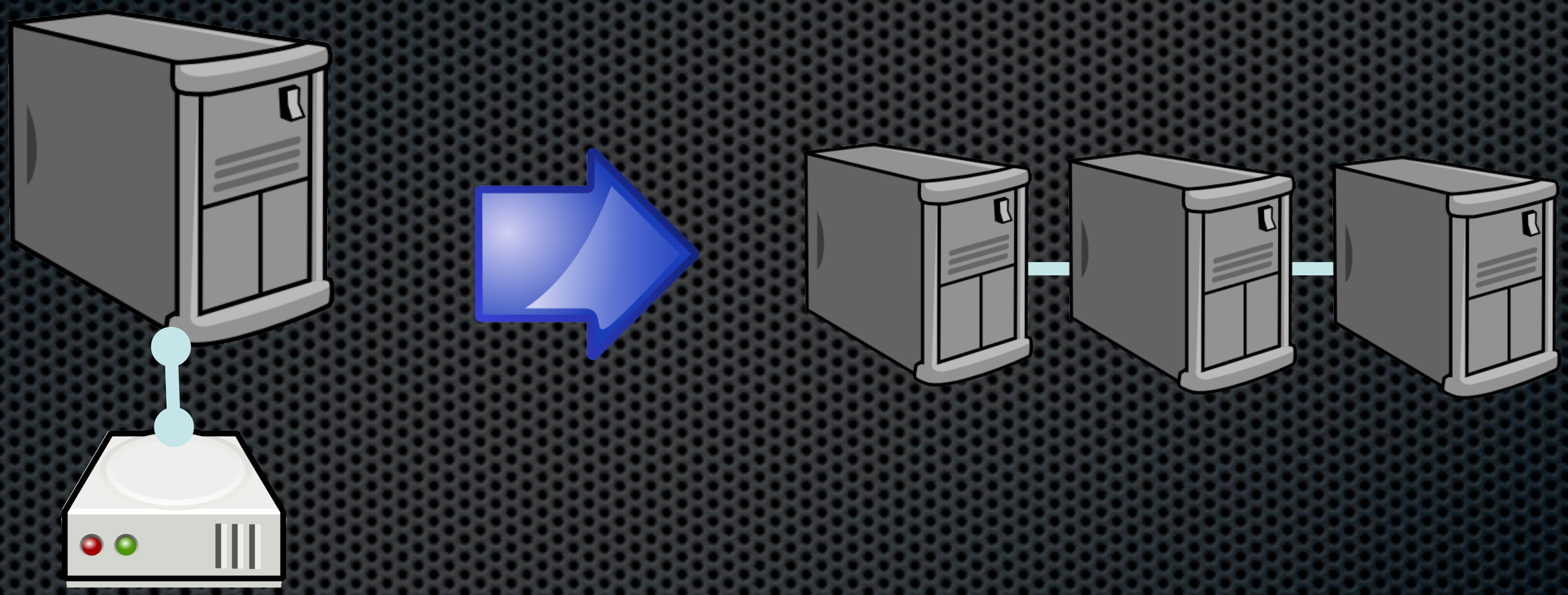
Improving Database Performance (3)

In Memory Databases



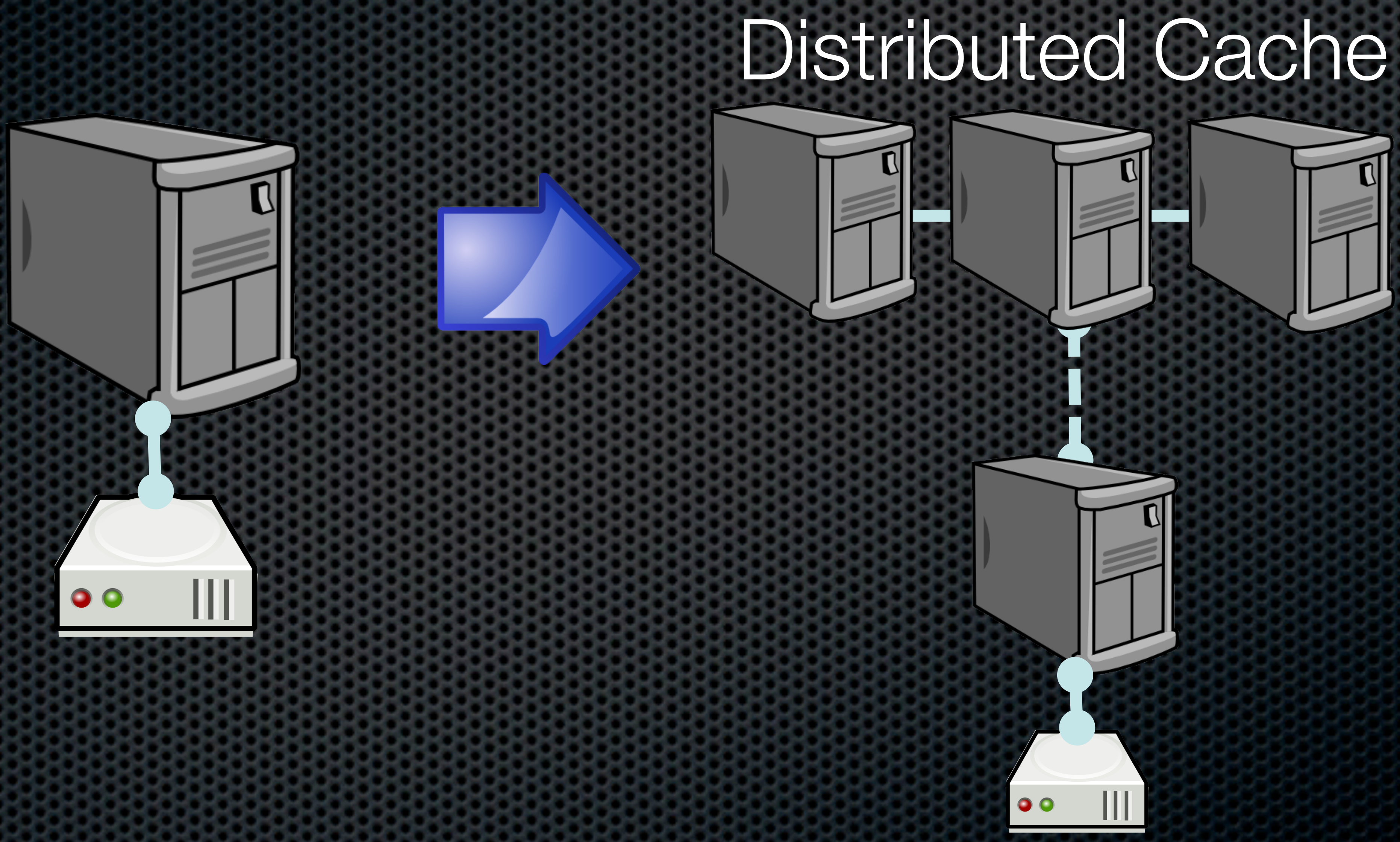
Improving Database Performance (4)

Distributed In Memory (Shared Nothing)

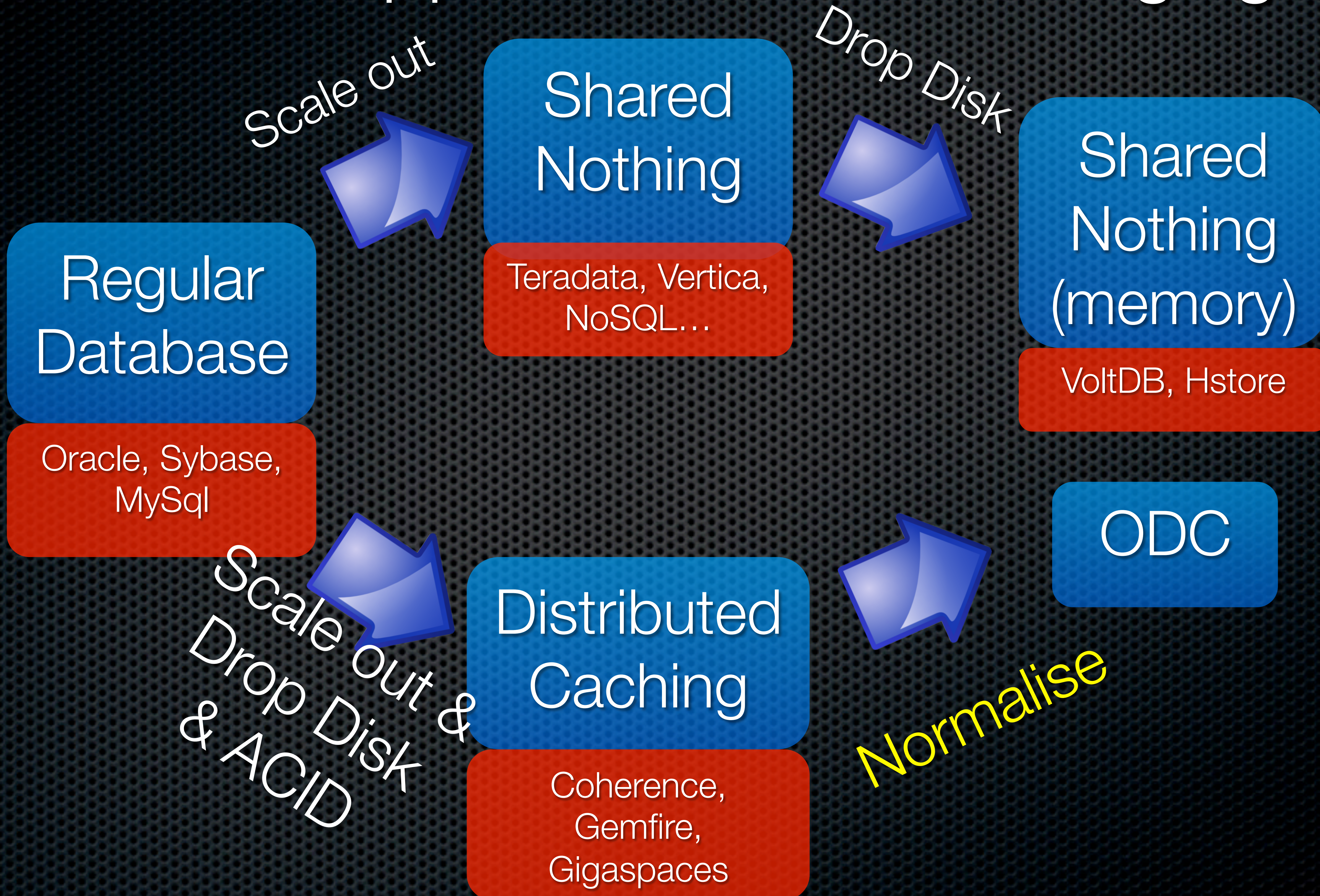


Improving Database Performance (5)

Distributed Caching



These approaches are converging

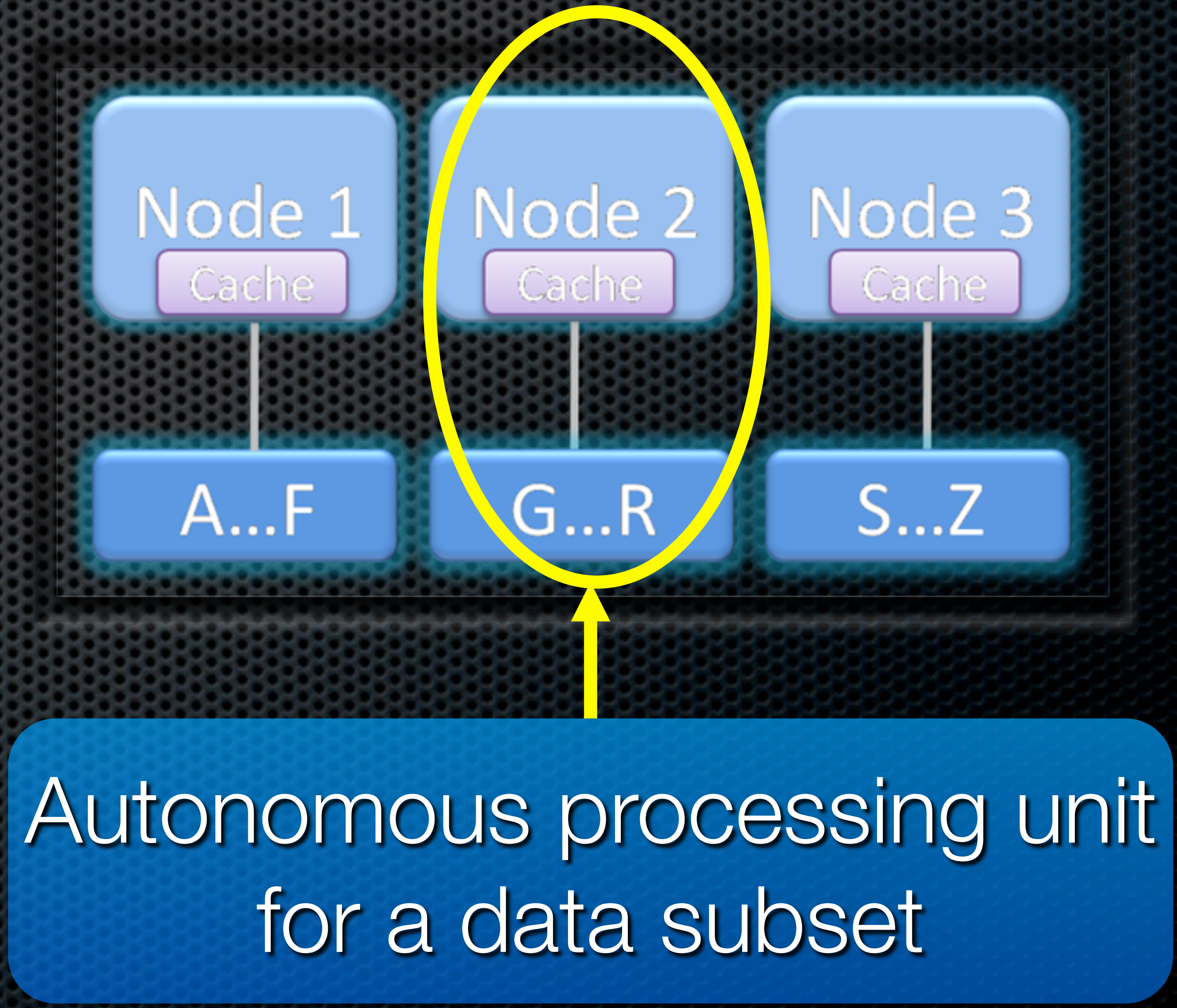


So how can we make a data store go even faster?

- Distributed Architecture
- Drop ACID: Simplify the Contract.
- Drop disk

(1) Distribution for Scalability: The Shared Nothing Architecture

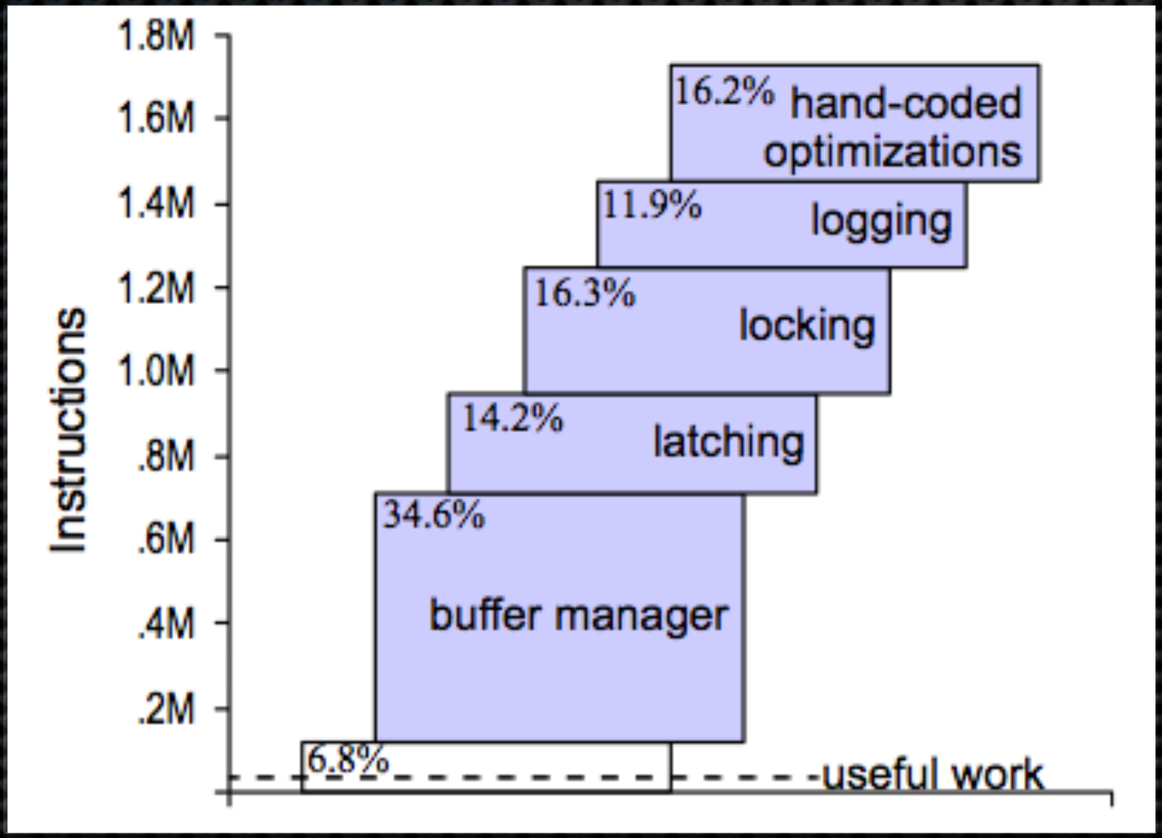
- Originated in 1990 (Gamma DB) but popularised by Teradata / BigTable / NoSQL
- Massive storage potential
- Massive scalability of processing
- Commodity hardware
- Limited by cross partition joins



(2) Simplifying the Contract

- For many users ACID is overkill.
- Implementing ACID in a distributed architecture has a significant affect on performance.
- NoSQL Movement: CouchDB, MongoDB,10gen, Basho, CouchOne, Cloudant, Cloudera, GoGrid, InfiniteGraph, Membase, Riptano, Scality….

Databases have huge operational overheads



Research with Shore DB indicates only 6.8% of instructions contribute to ‘useful work’

Taken from “OLTP Through the Looking Glass, and What We Found There” Harizopoulos et al

(3) Memory is 100x faster than disk

ms μs ns ps

1MB Disk/Network

1MB Main Memory

0.000,000,000,000

Cross Continental Round Trip

Cross Network Round Trip

Main Memory Ref

L2 Cache Ref

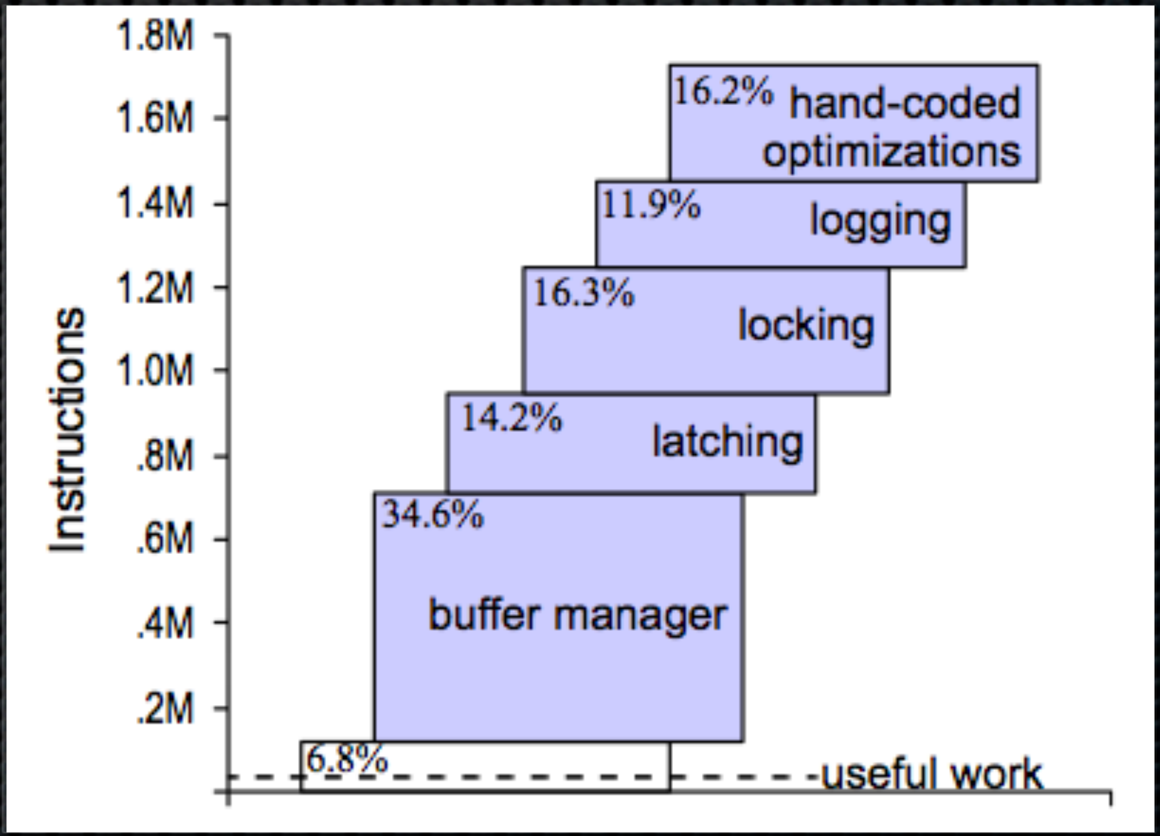
L1 Cache Ref

* L1 ref is about 2 clock cycles or 0.7ns. This is the time it takes light to travel 20cm

Avoid all that overhead

RAM means:

- No IO
- Single Threaded
  ⇒No locking / latching
- Rapid aggregation etc
- Query plans become less important



We were keen to leverage these three factors in building the ODC

Distribution

Simplify the contract

Memory Only

What is the ODC?

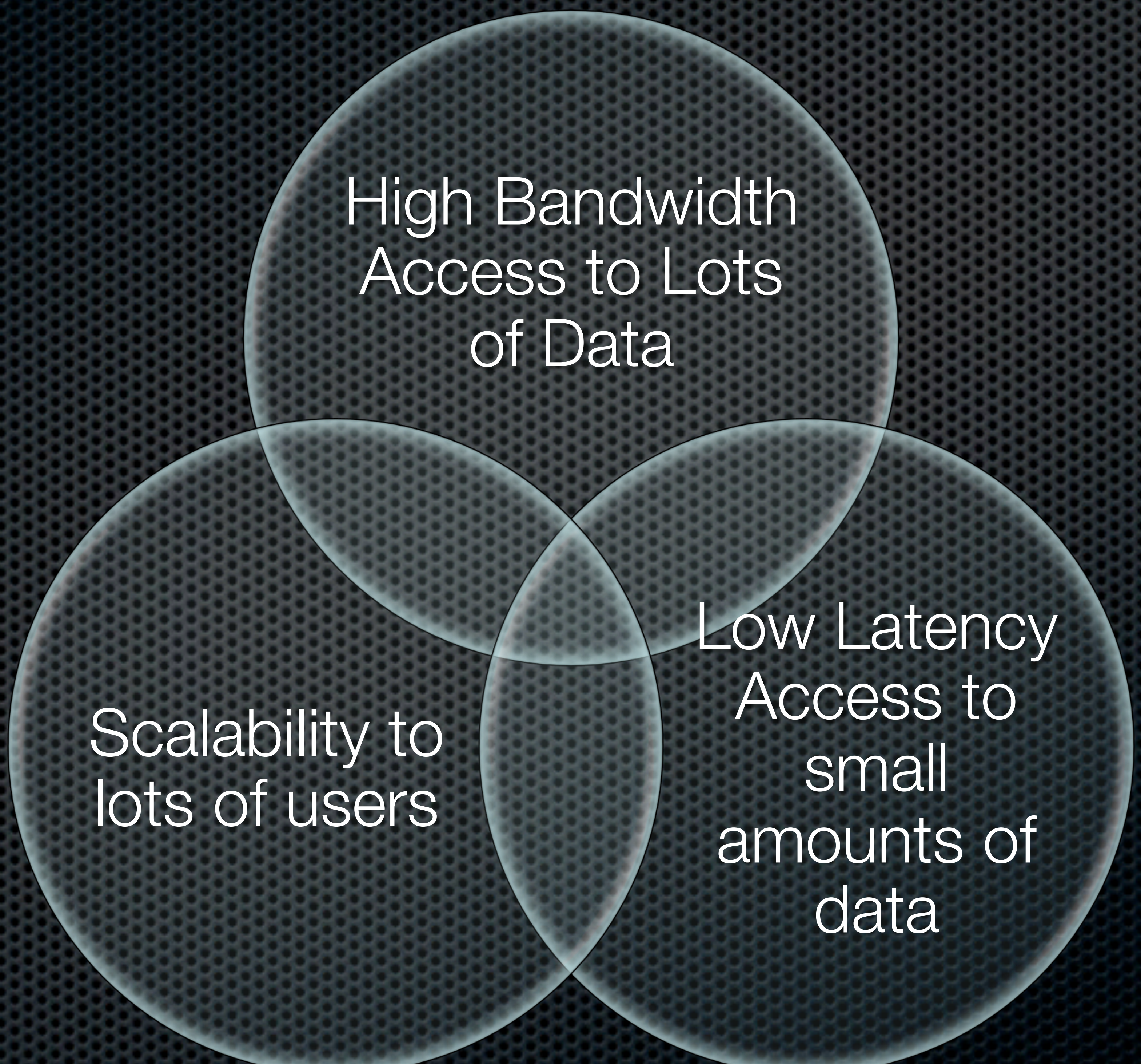
Highly distributed, in memory, normalised data store designed for scalable data access and processing.

The Concept

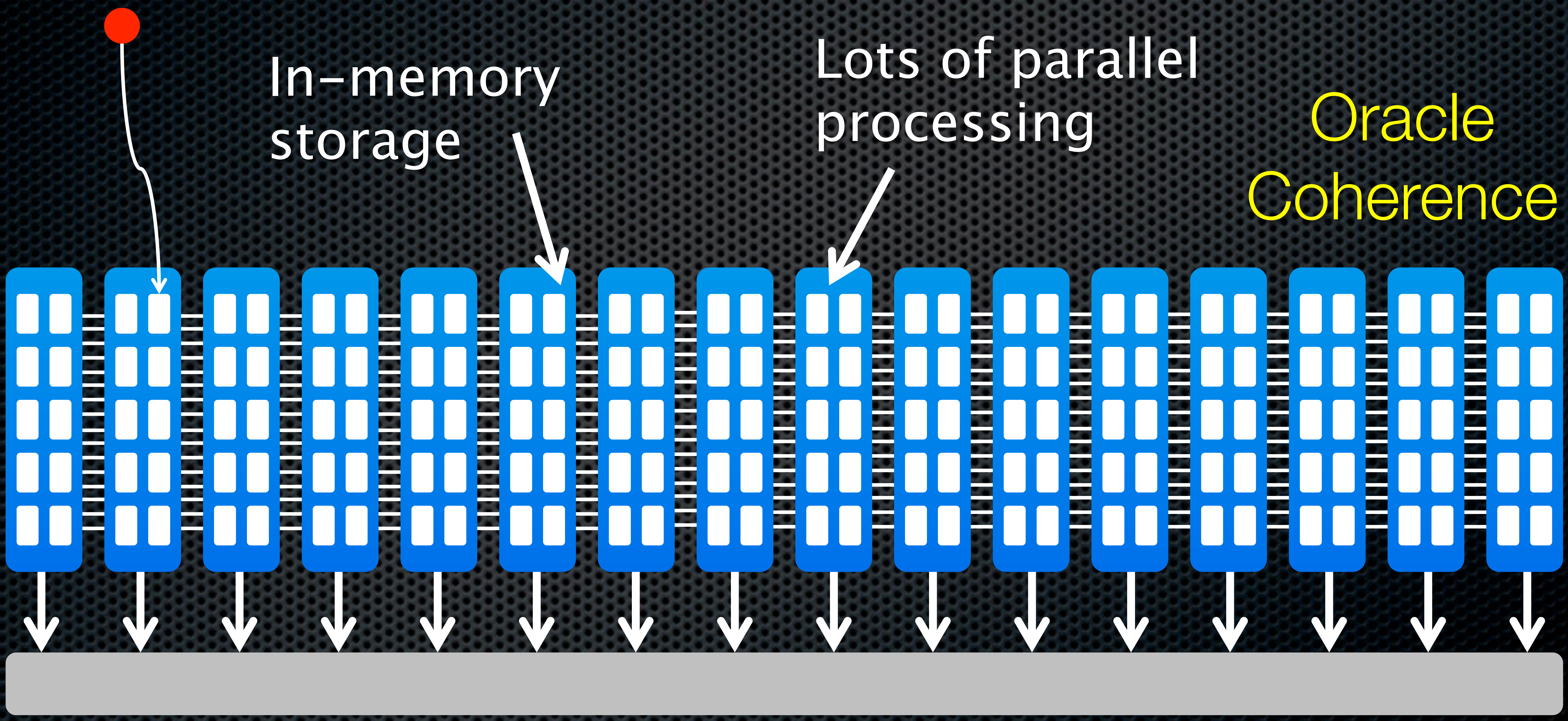
Originating from Scott Marcar's concept of a central brain within the bank:

"The copying of data lies at the route of many of the bank's problems. By supplying a single real-time view that all systems can interface with we remove the need for reconciliation and promote the concept of truly shared services" - Scott Marcar (Head of Risk and Finance Technology)

This is quite tricky problem

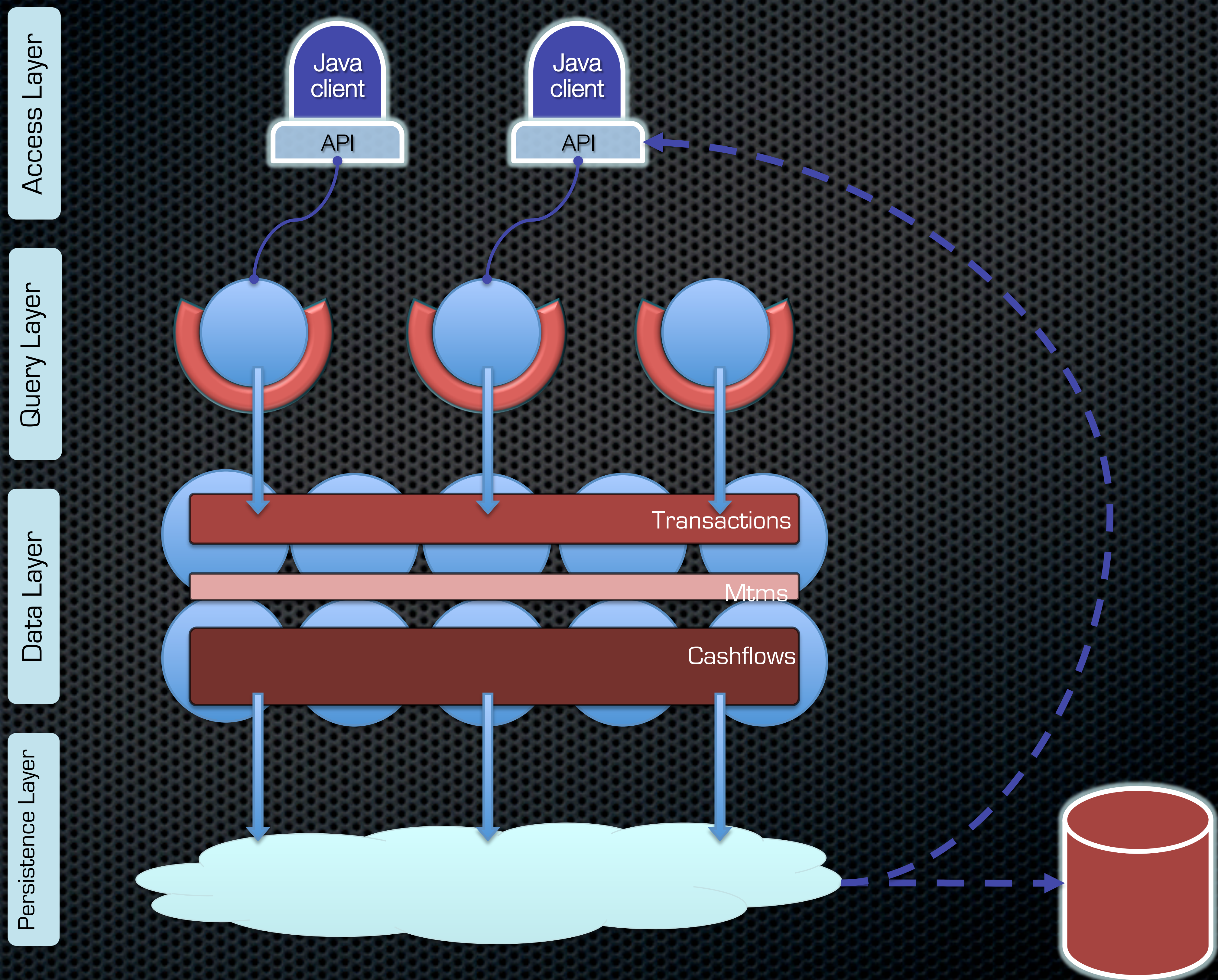


ODC Data Grid: Highly Distributed Physical Architecture



Messaging (Topic Based) as a system of record (persistence)

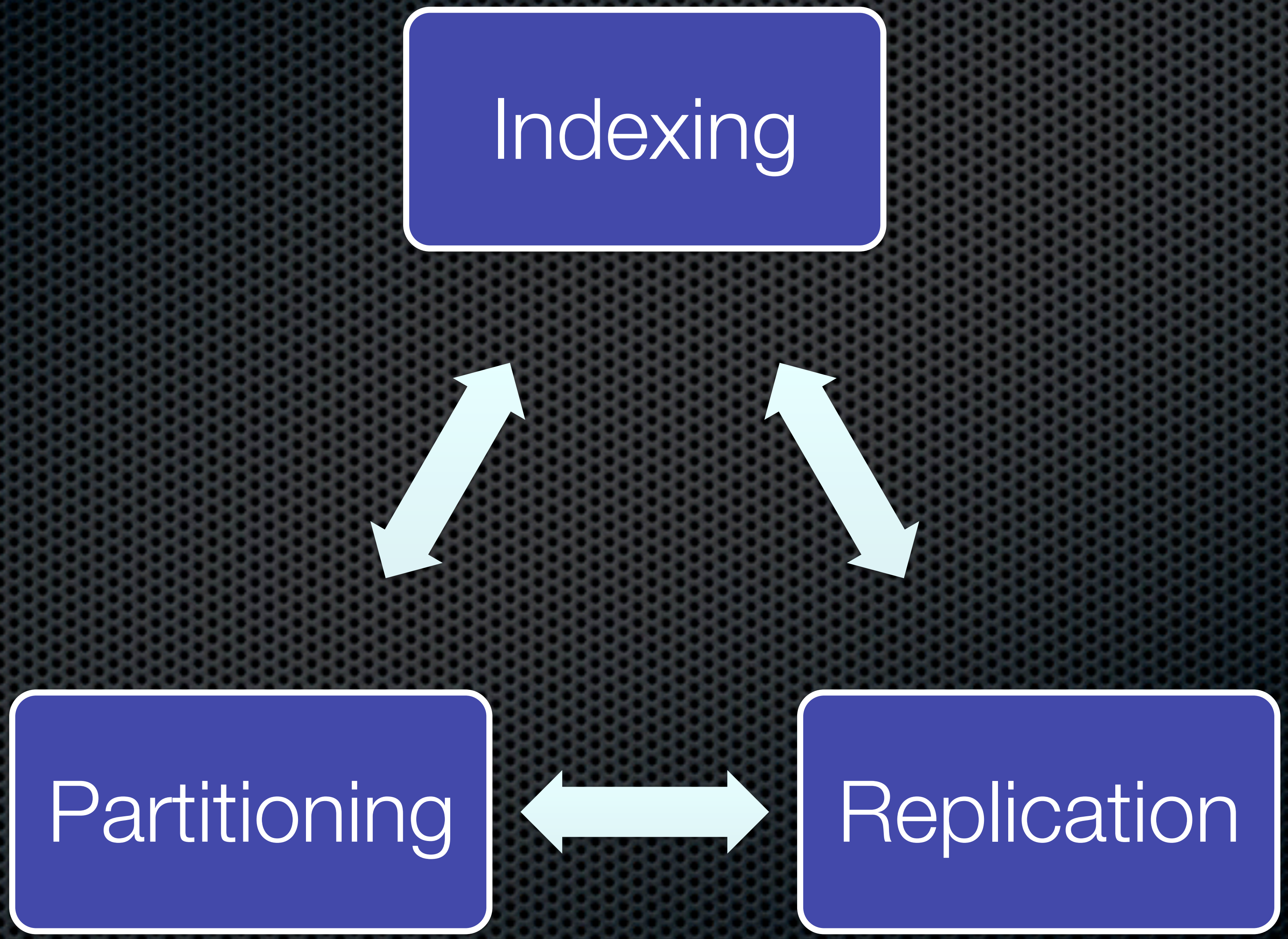
The Layers



Access Layer
Query Layer
Data Layer
Persistence Layer
Java client
API
Java client
API
Transactions
Mtms
Cashflows

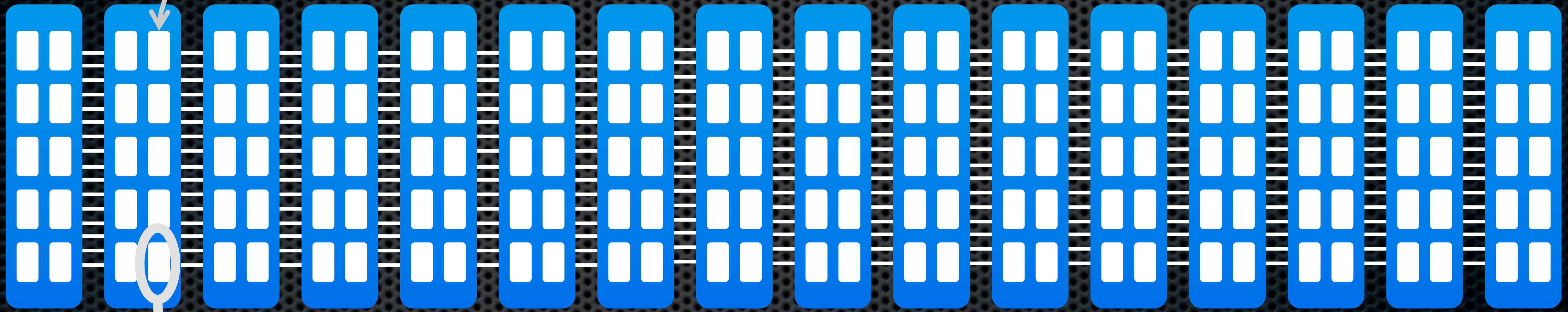
But unlike most caches the ODC is *Normalised*

Three Tools of Distributed Data Architecture



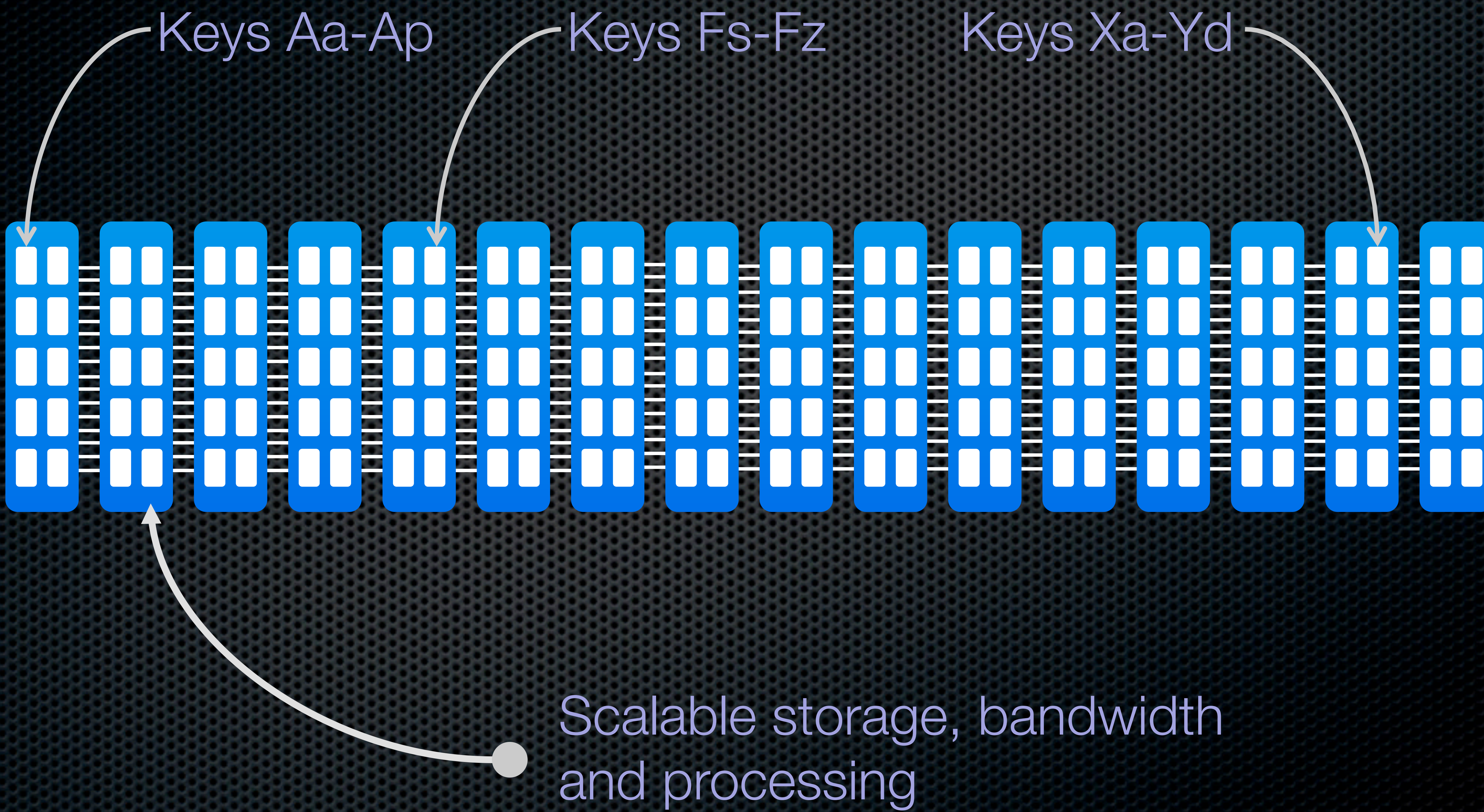
For speed, replication is best

Wherever you go the data will be there

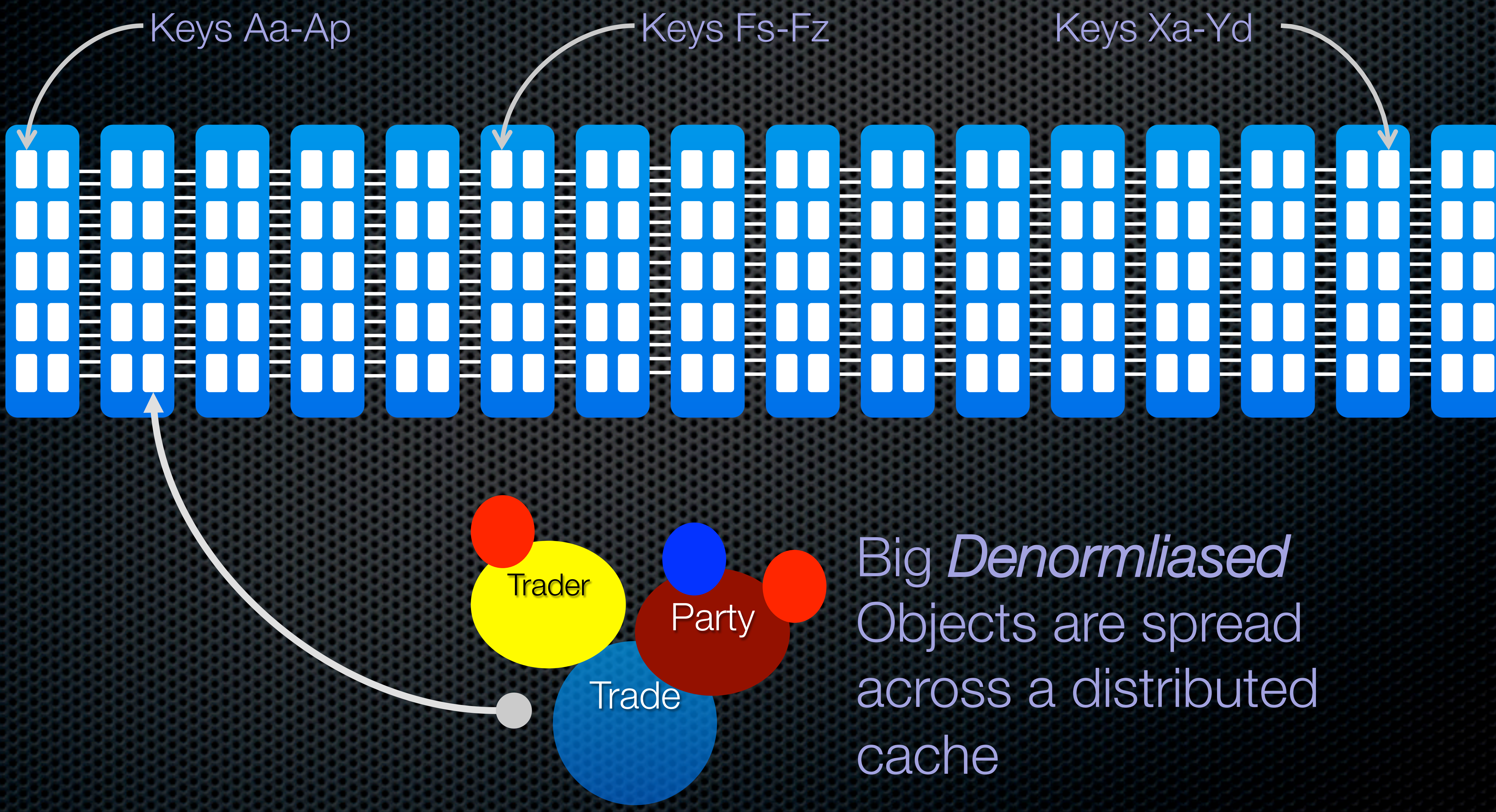


But your storage is limited by the memory on a node

For scalability, partitioning is best



Traditional Distributed Caching Approach



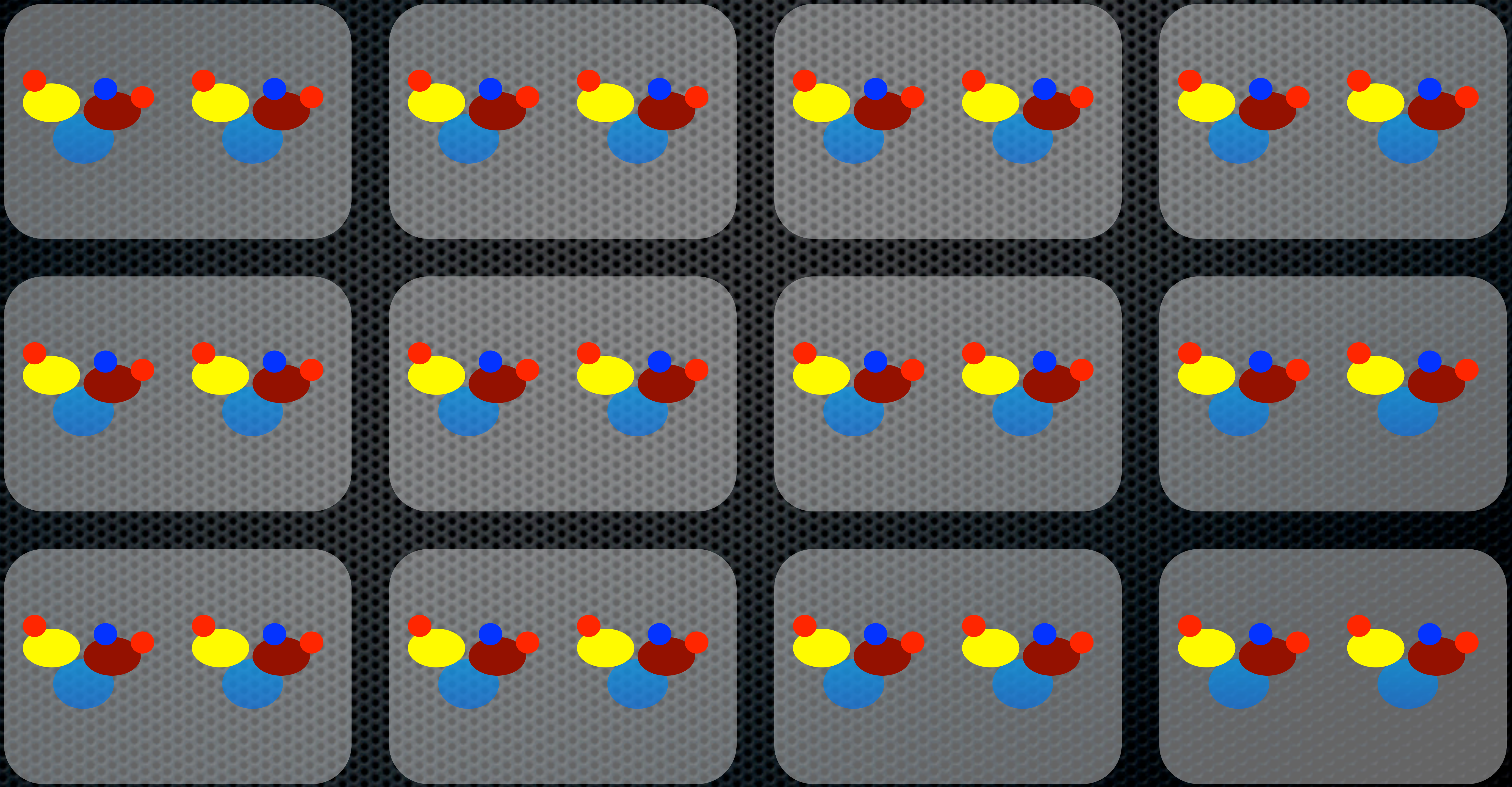
Big *Denormliased* Objects are spread across a distributed cache

But we believe a data store needs to be more than this: it needs to be normalised!

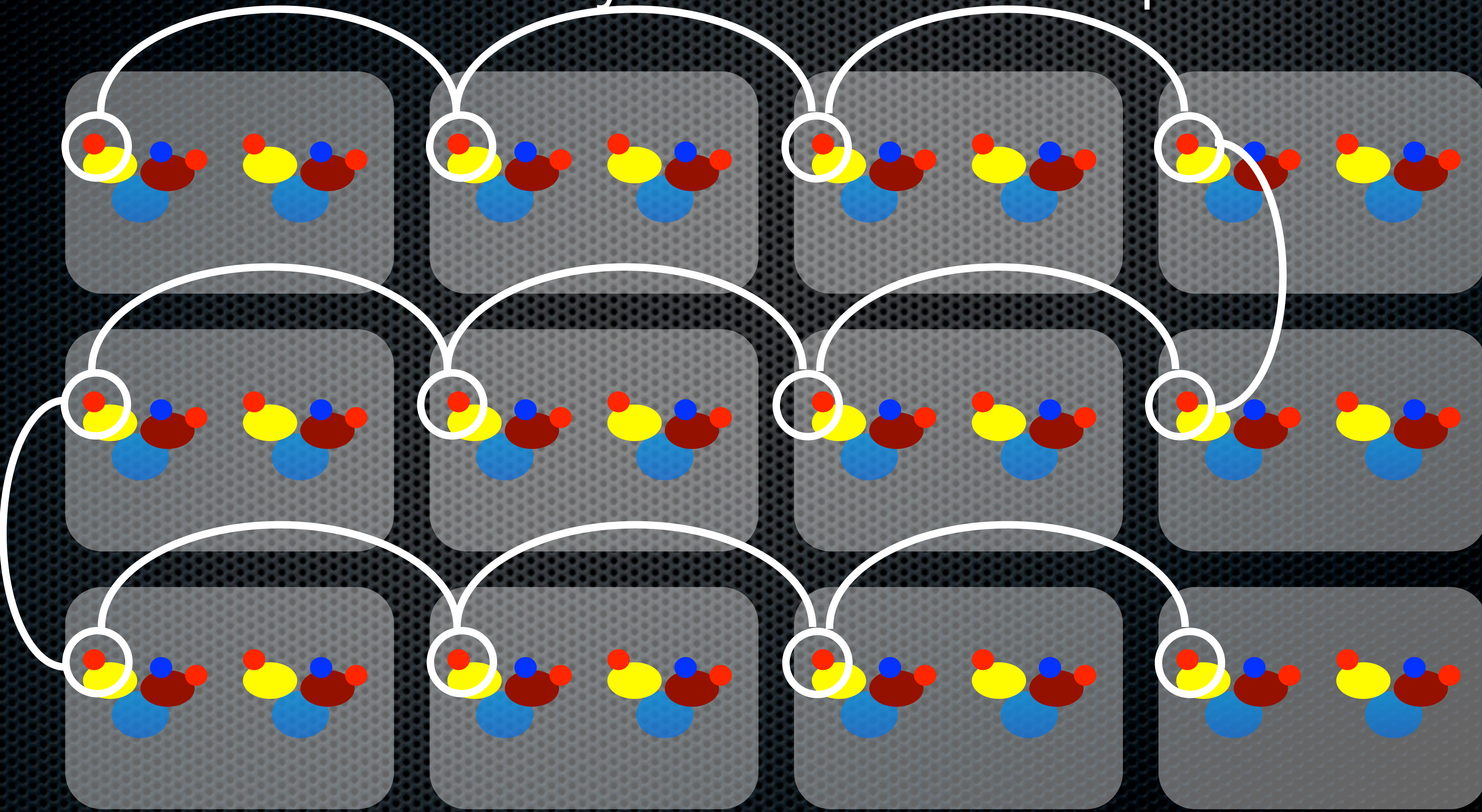
So why is that?

Surely denormalisation is going to be faster?

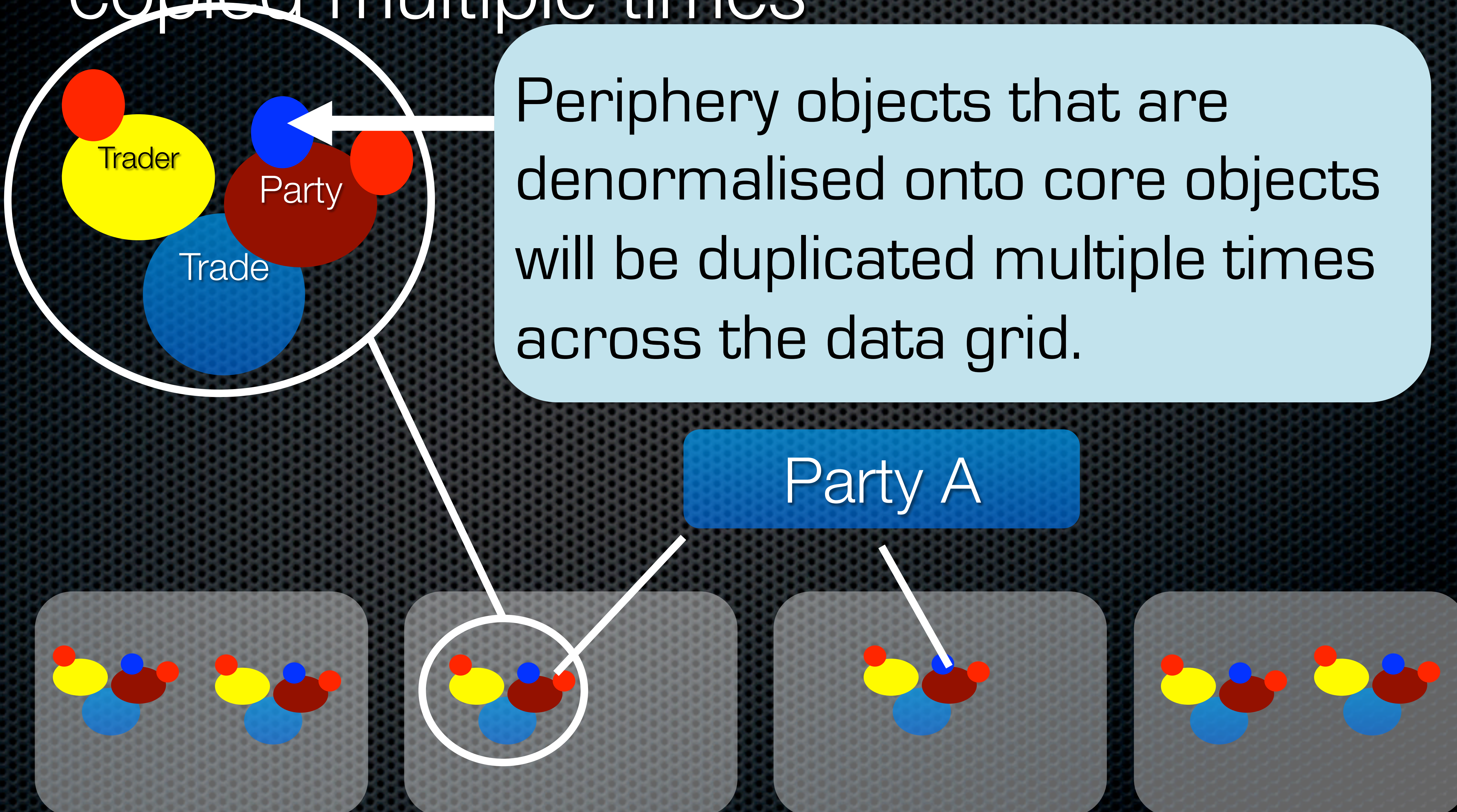
Denormalisation means replicating parts of your object model



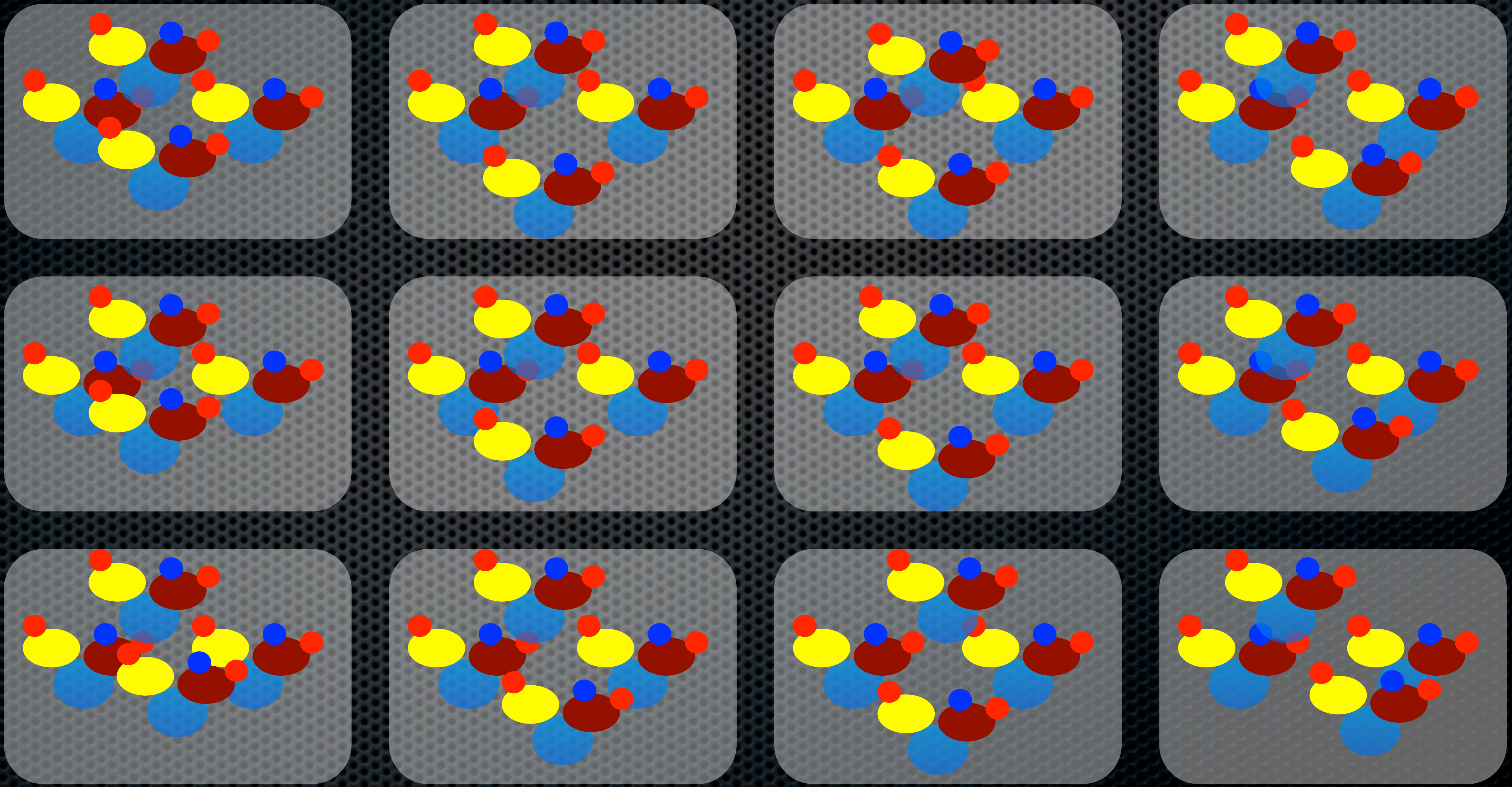
...and that means managing consistency over lots of copies



... as parts of the object graph will be copied multiple times



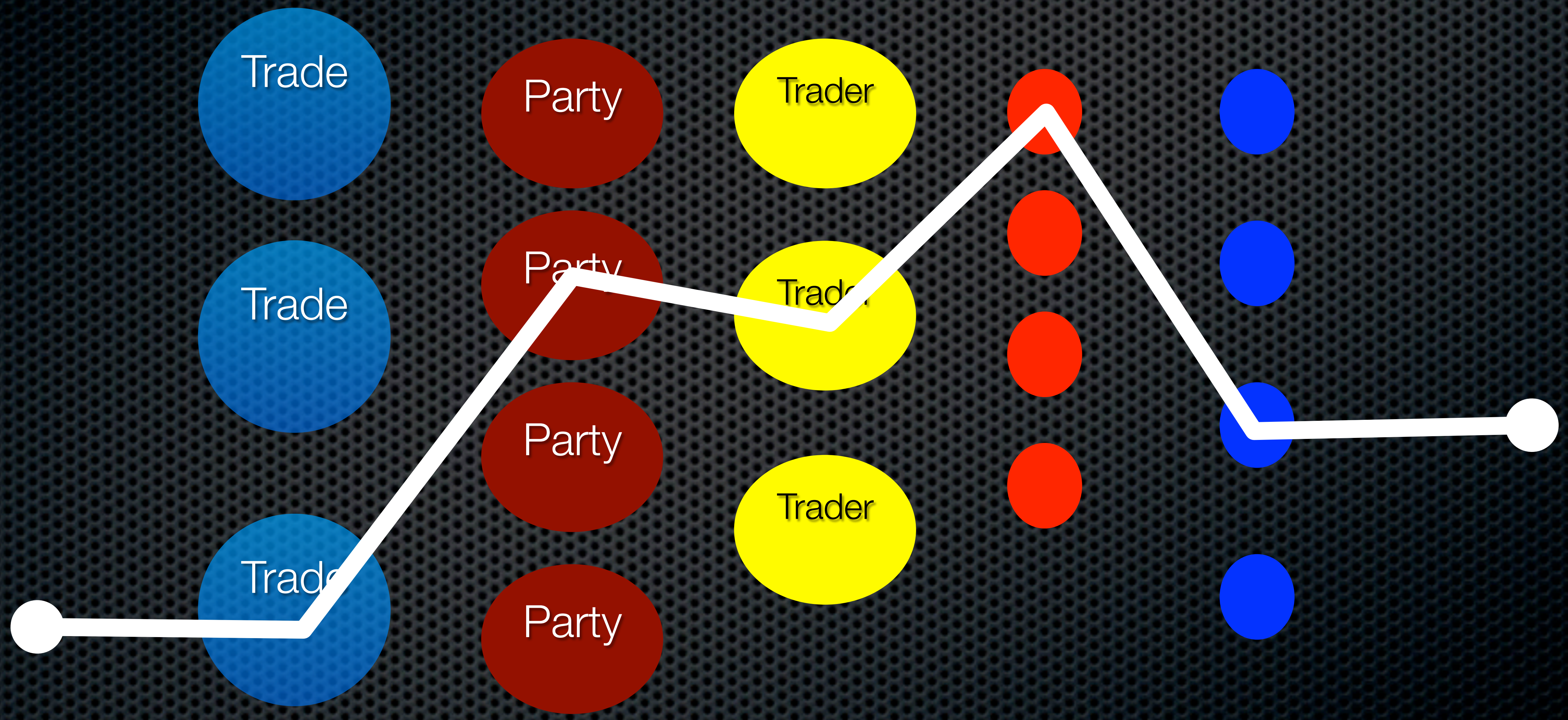
…and all the duplication means you run out of space really quickly



Spaces issues are exaggerated further when data is versioned

…and you need versioning to do MVCC

And reconstituting a previous time slice becomes very difficult.



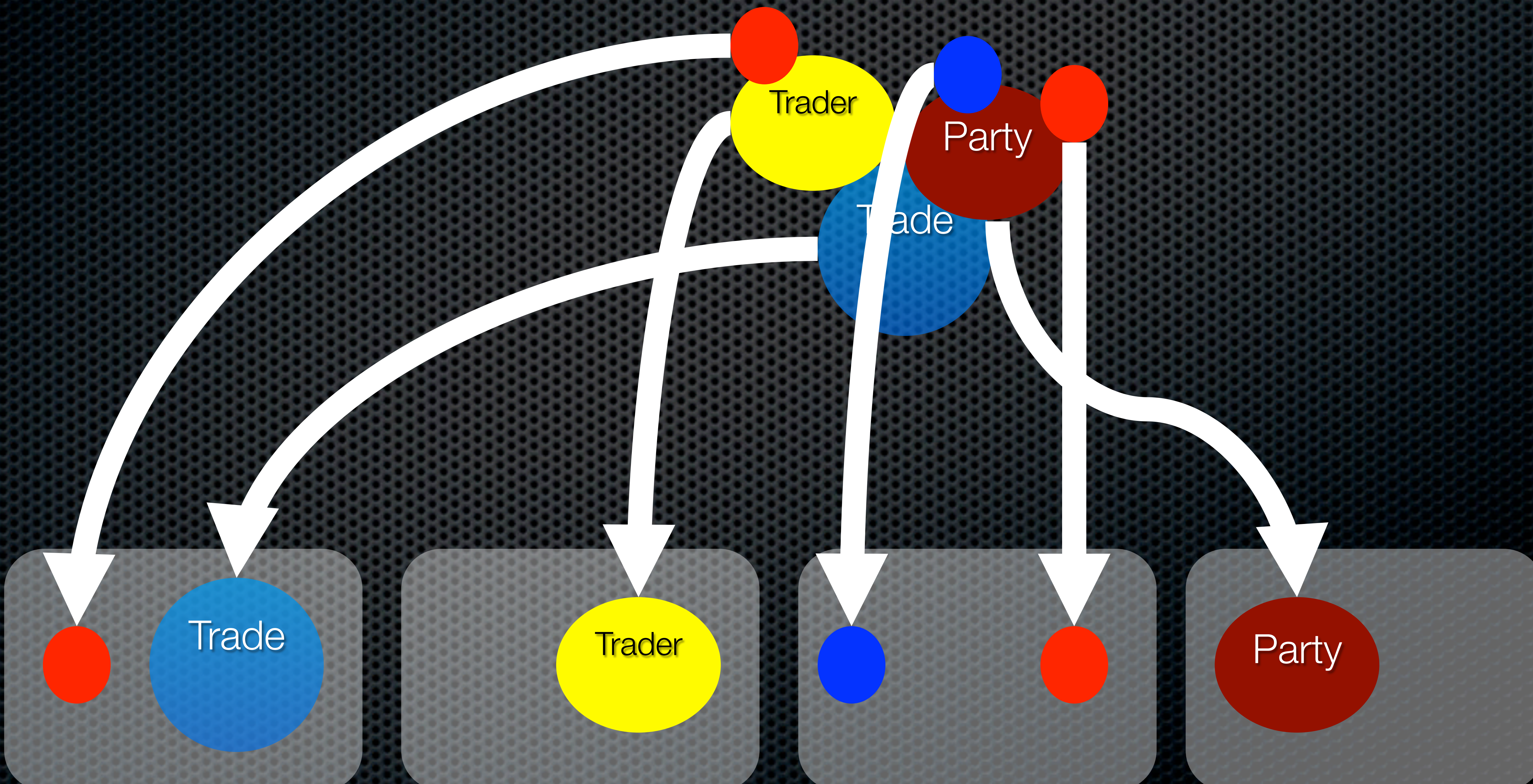
Why Normalisation?

- Easy to change data (no distributed locks / transactions)
- Better use of memory.
- Facilitates Versioning
- And MVCC/Bi-temporal

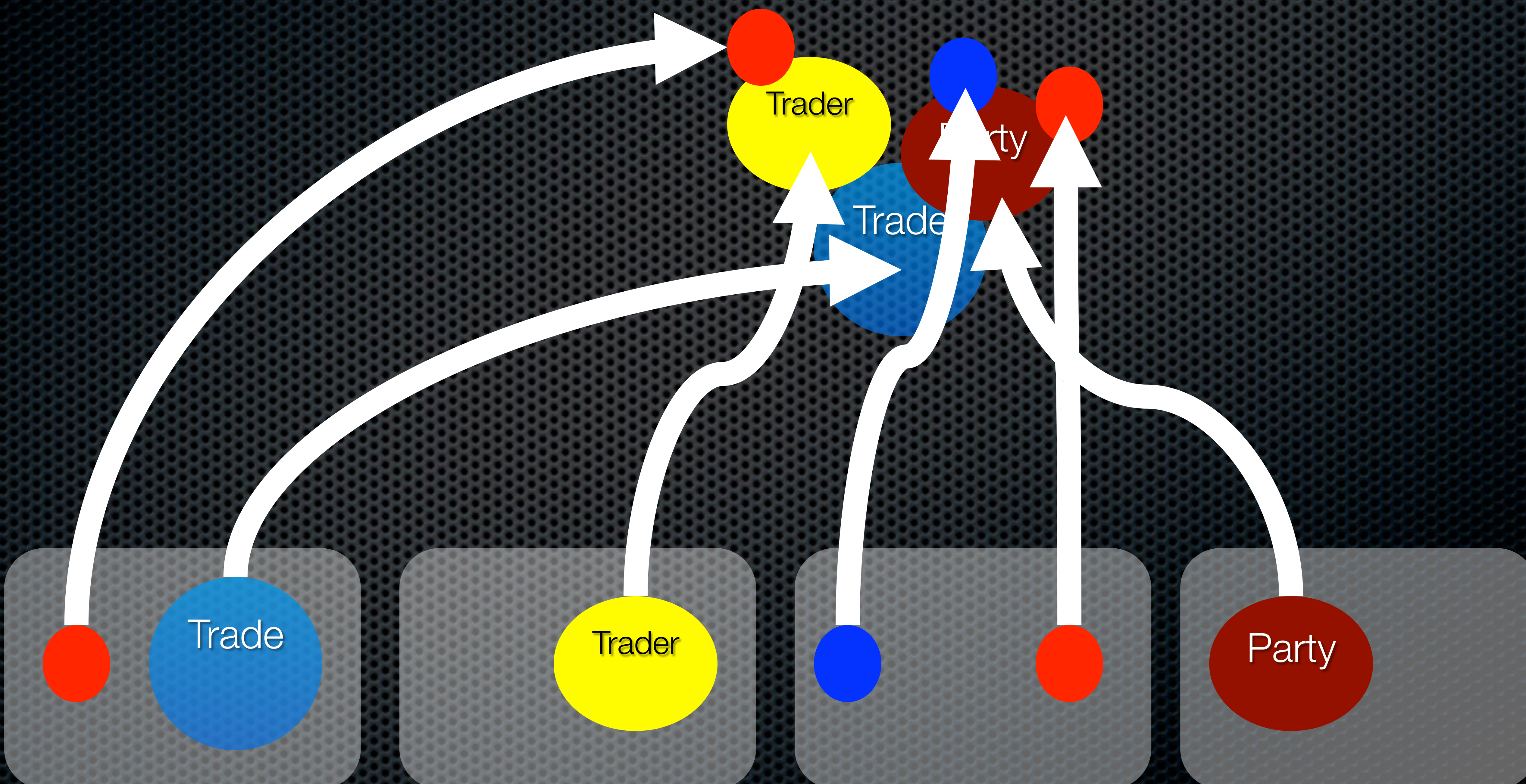
OK, OK, lets normalise our data then. What does that mean?

We decompose our domain model and hold each object separately

This means the object graph will be split across multiple machines.

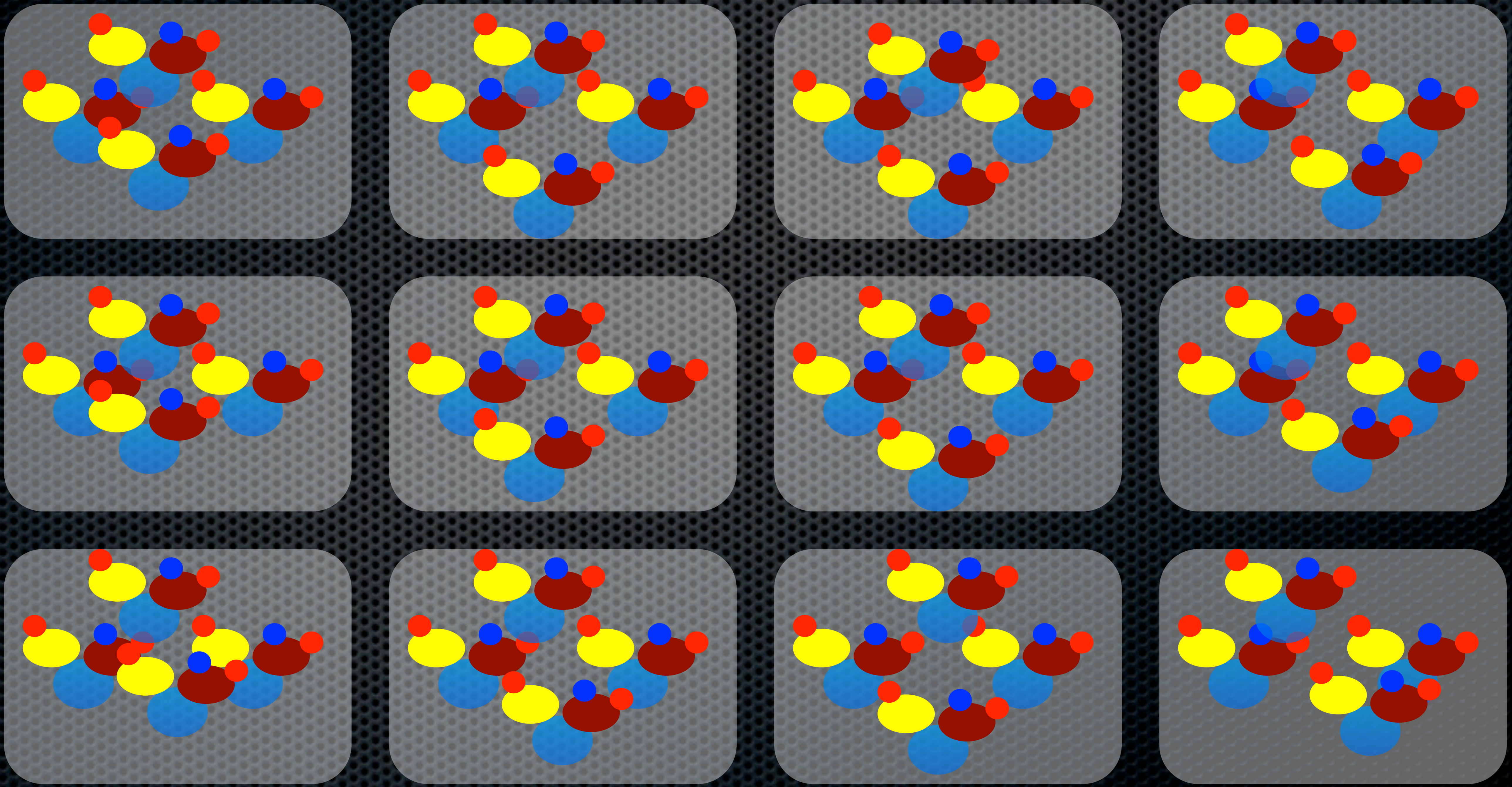


Binding them back together involves a "distributed join" => Lots of network hops



It's going to be slow…

Whereas the denormalised model the join is already done



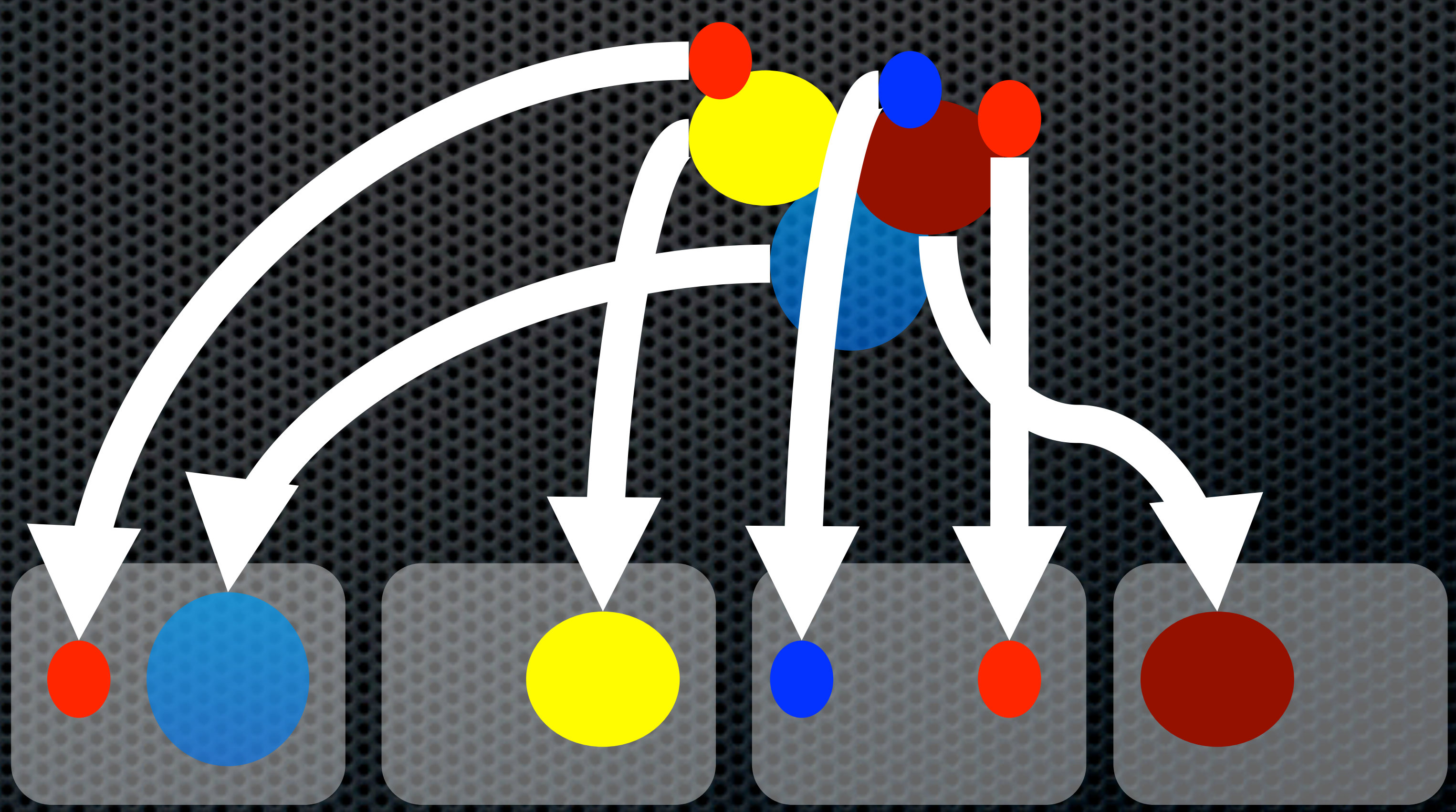
Hence Denormalisation is *FAST!*

(for reads)

So what we want is the advantages of a normalised store at the speed of a denormalised one!

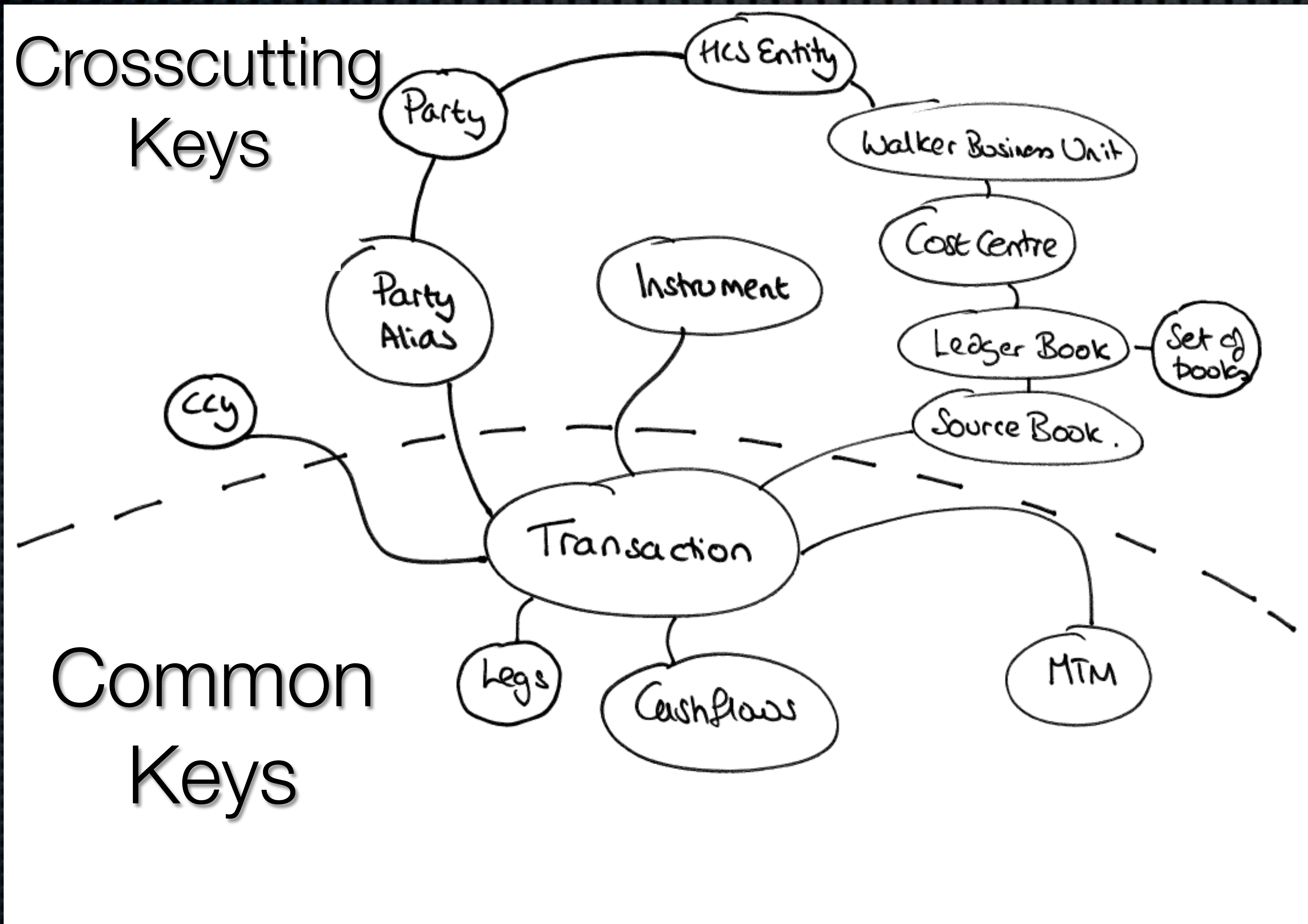
This is what the ODC is all about!

Looking more closely: Why does normalisation mean we have to be spread data around the cluster. Why can't we hold it all together?

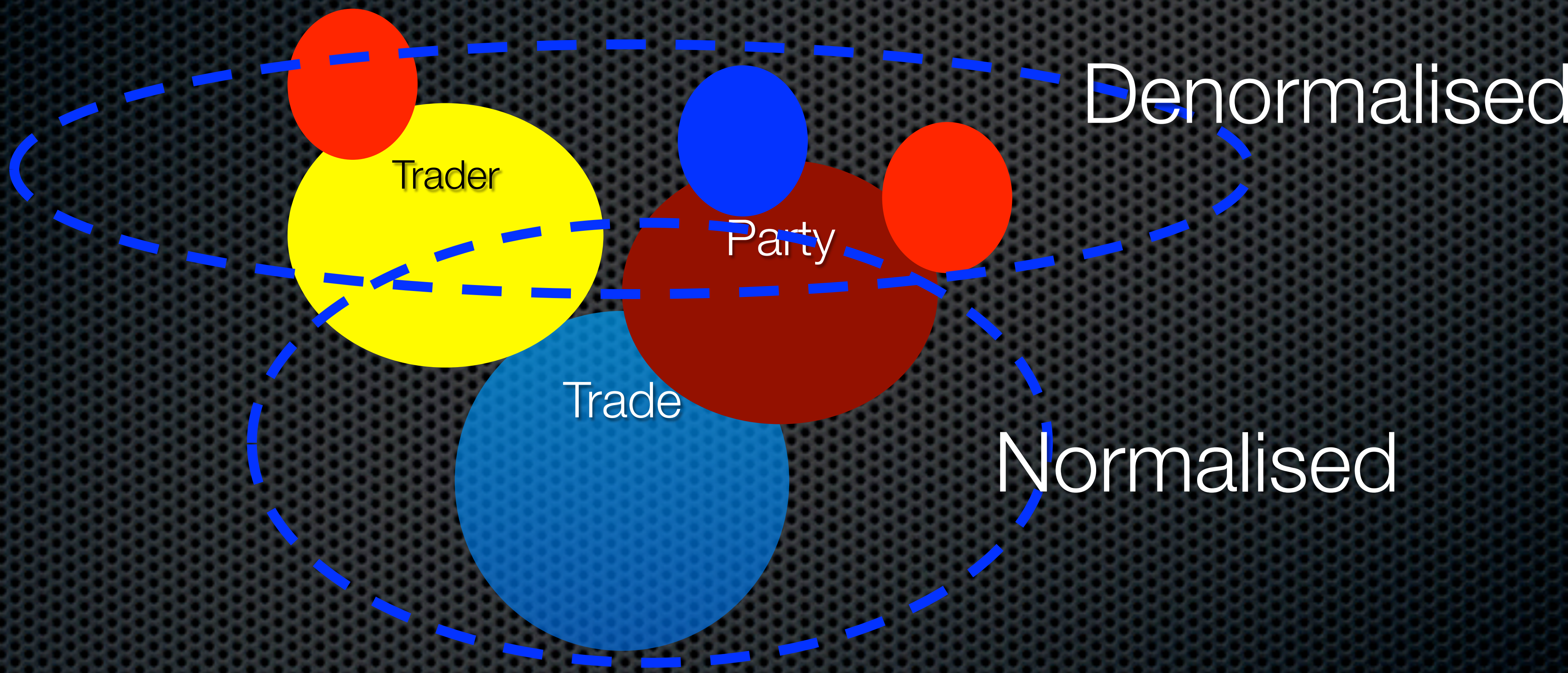


It’s all about the keys

We can collocate data with common keys but if they crosscut the only way to collocate is to replicate

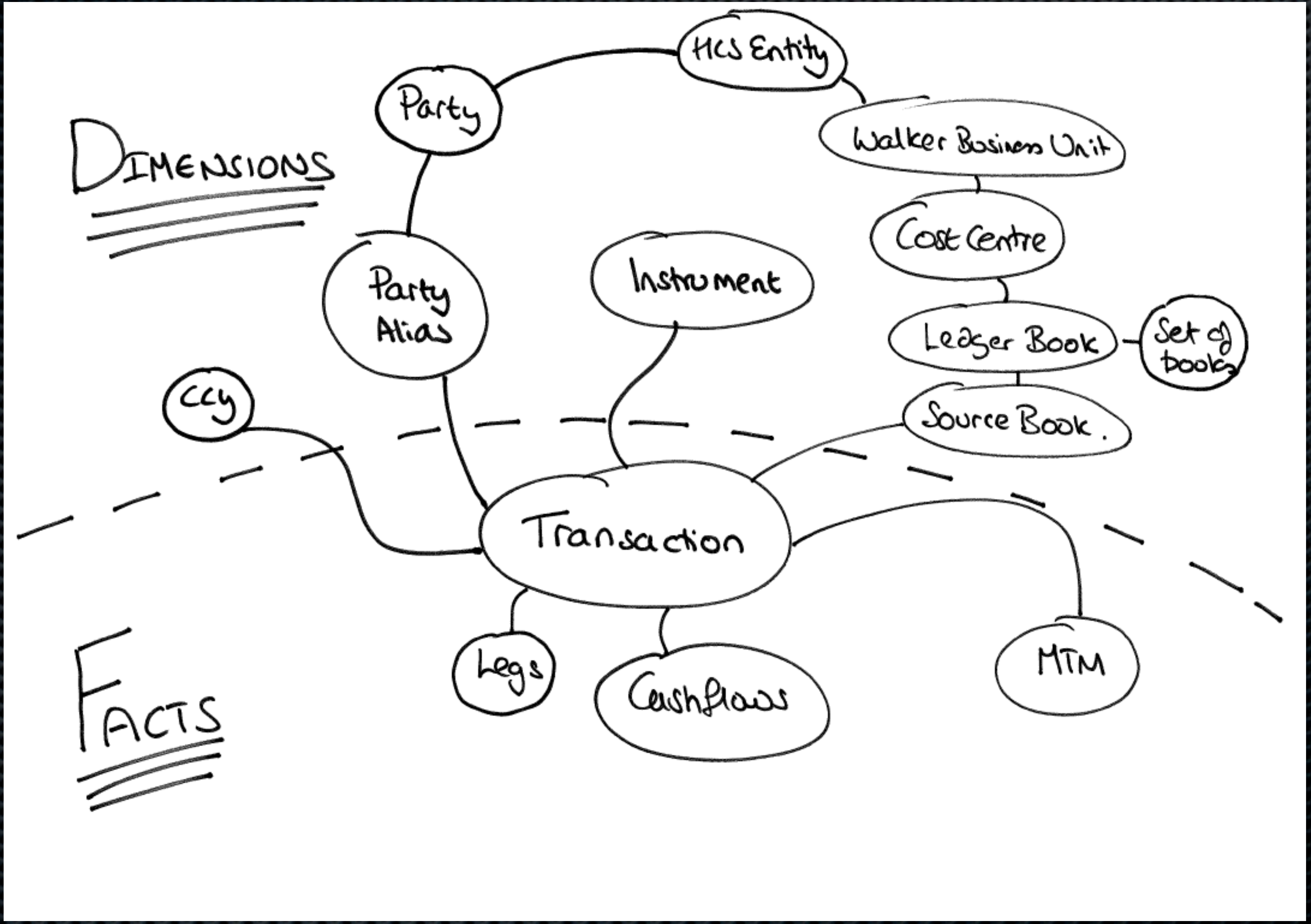


We tackle this problem with a hybrid model:



We adapt the concept of a Snowflake Schema.

Taking the concept of *Facts* and *Dimensions*



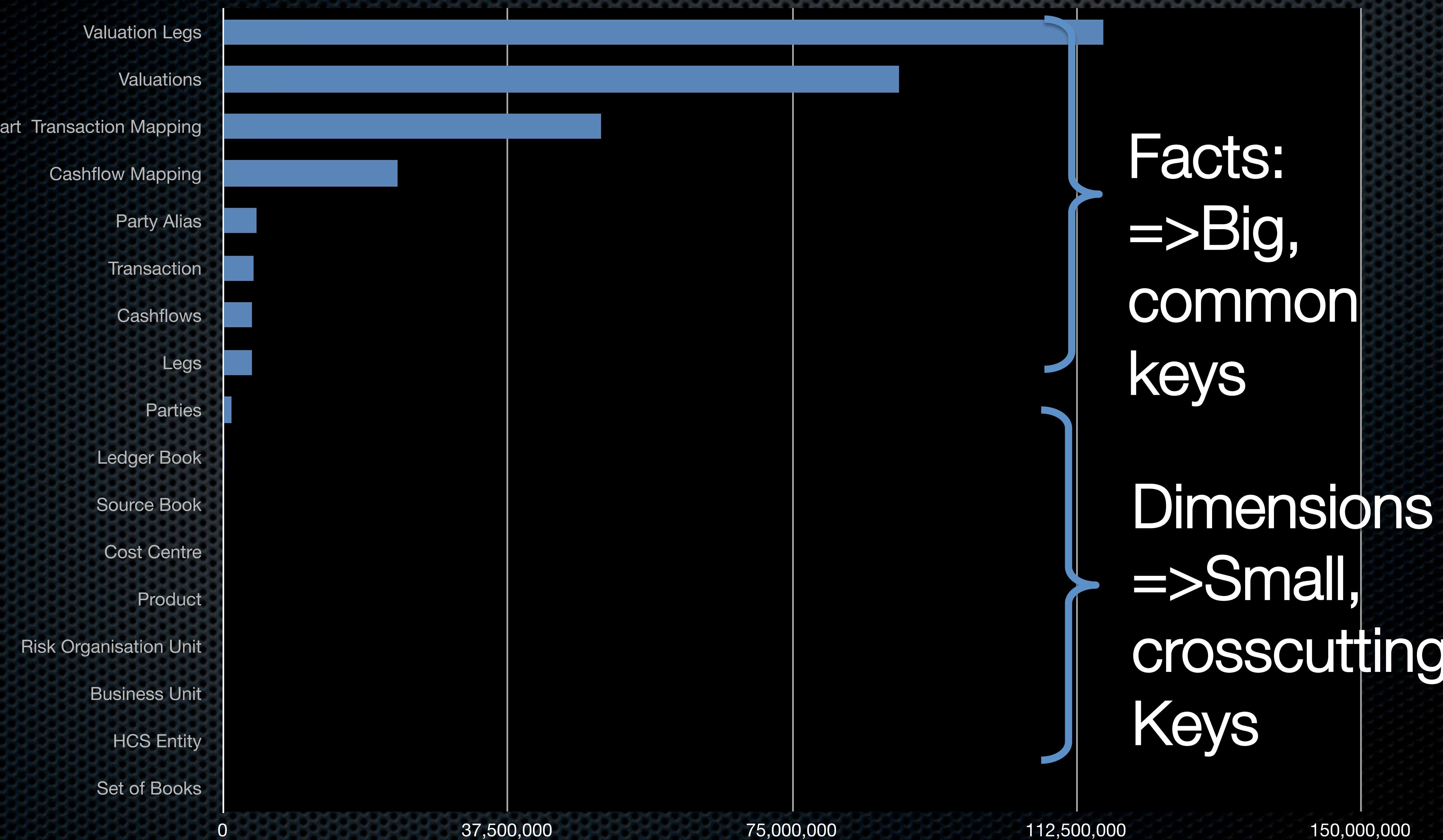
Everything starts from a Core Fact
(Trades for us)

Facts are Big, dimensions are small

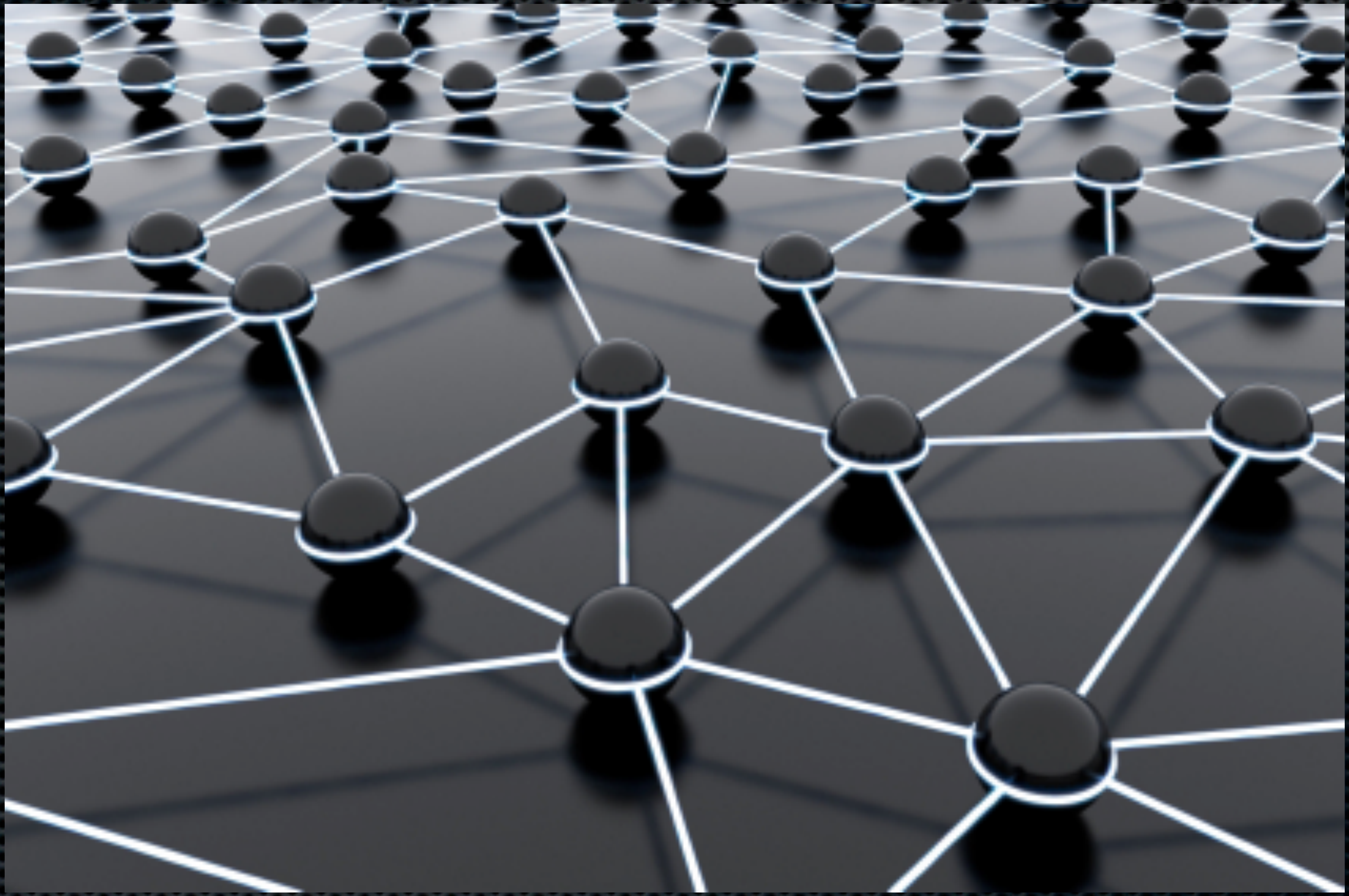
Facts have one key

Dimensions have many (crosscutting) keys

Looking at the data:

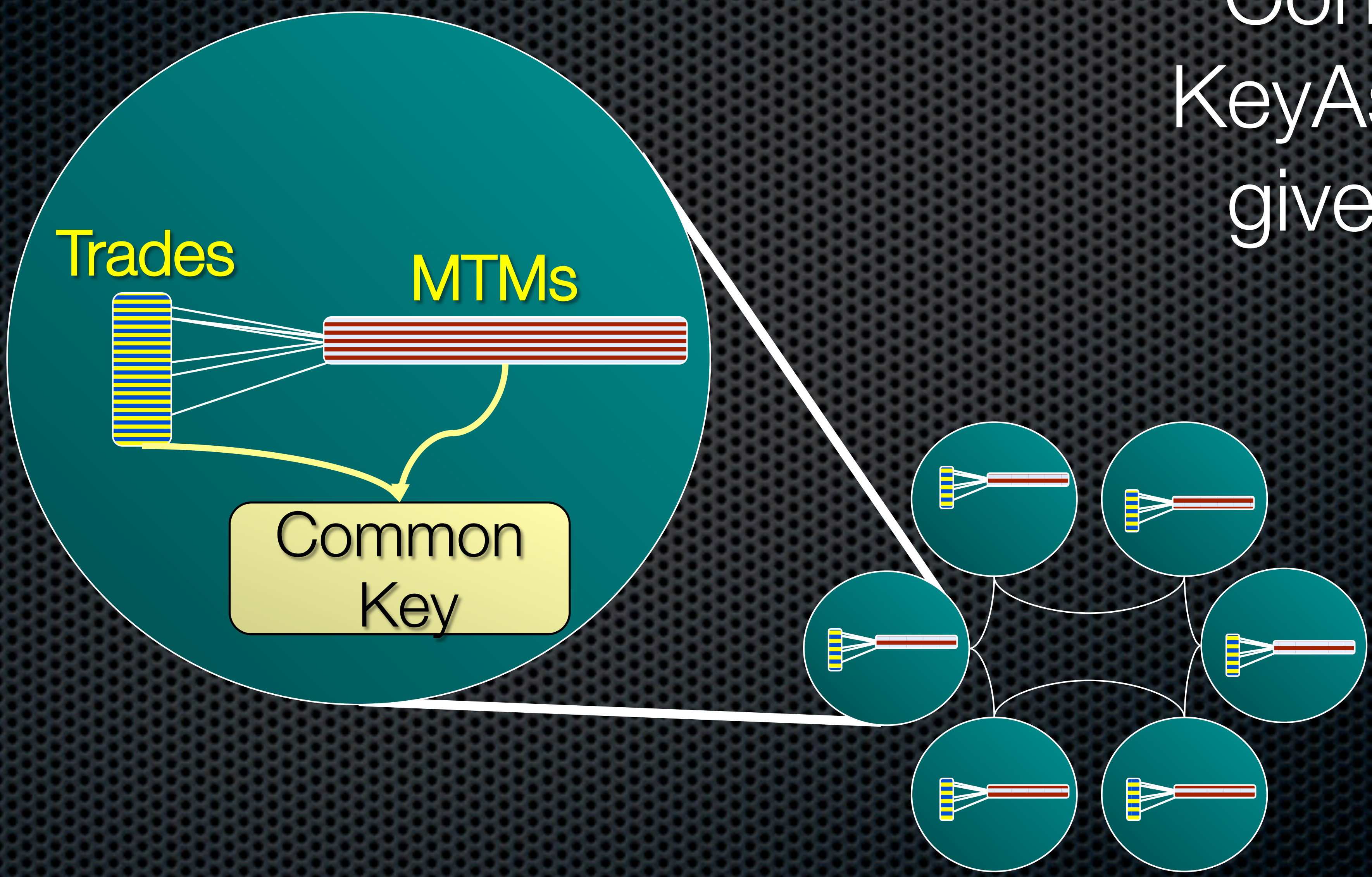


We remember we are a grid. We should avoid the distributed join.

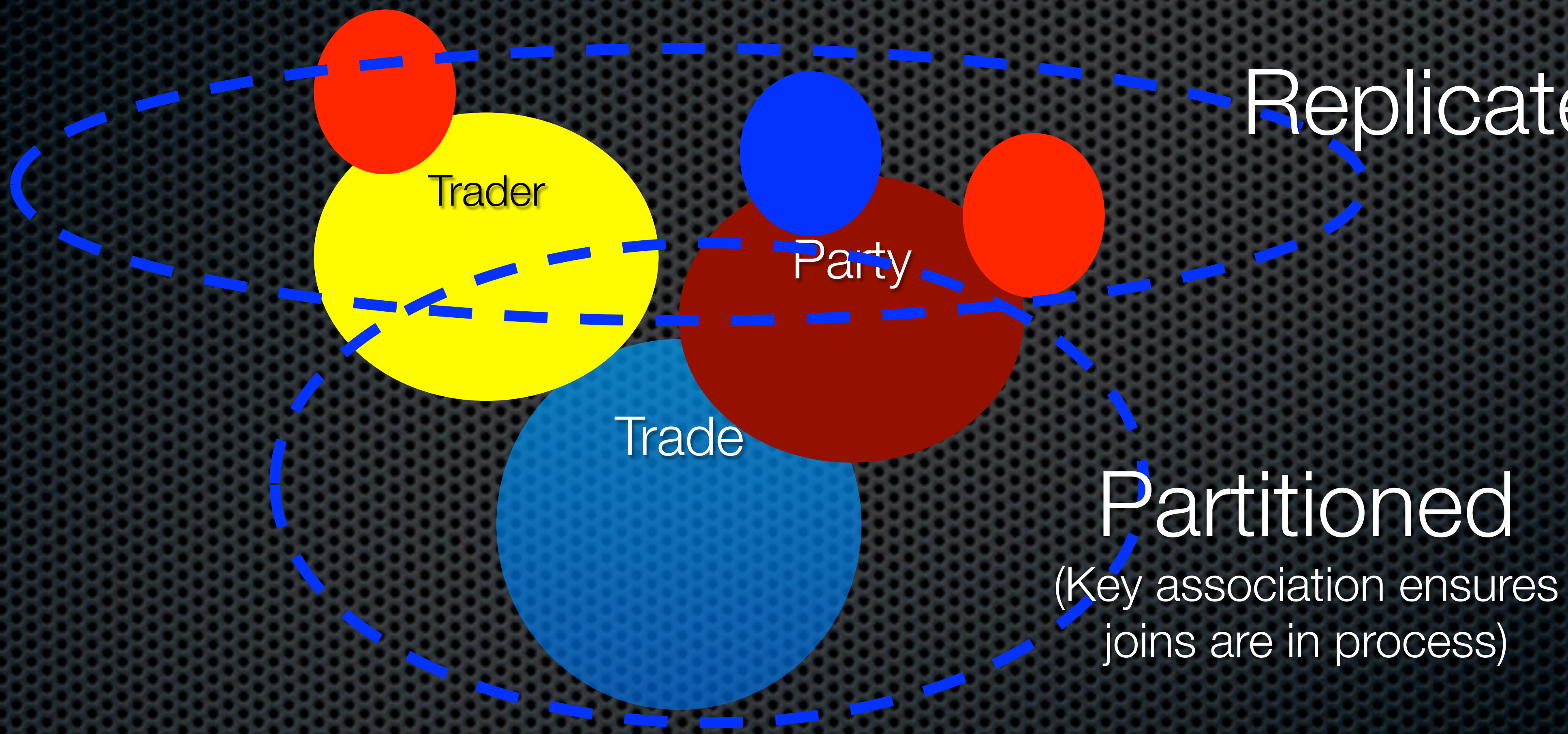


… so we only want to 'join' data that is in the same process

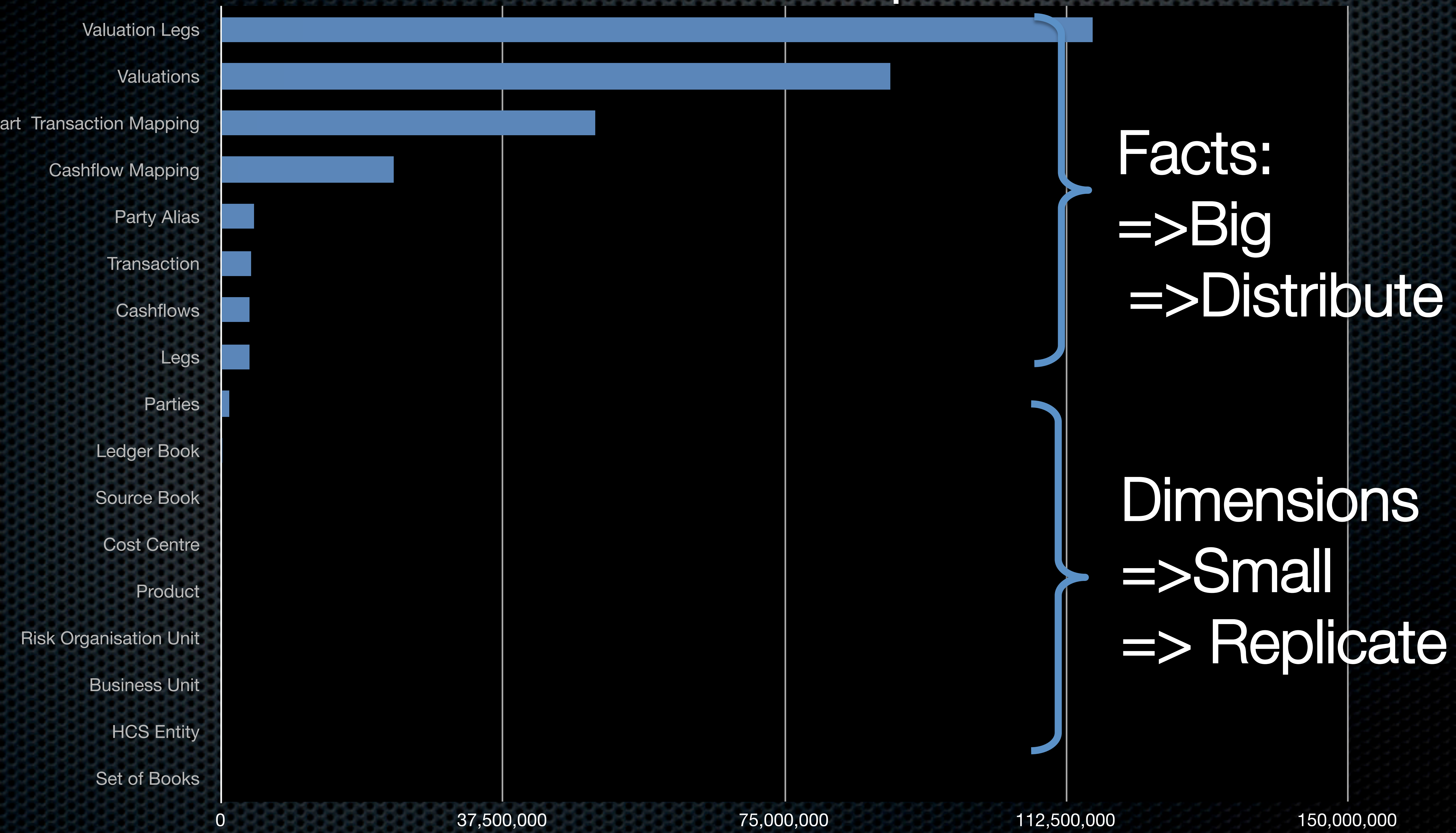
Coherence's KeyAssociation gives us this



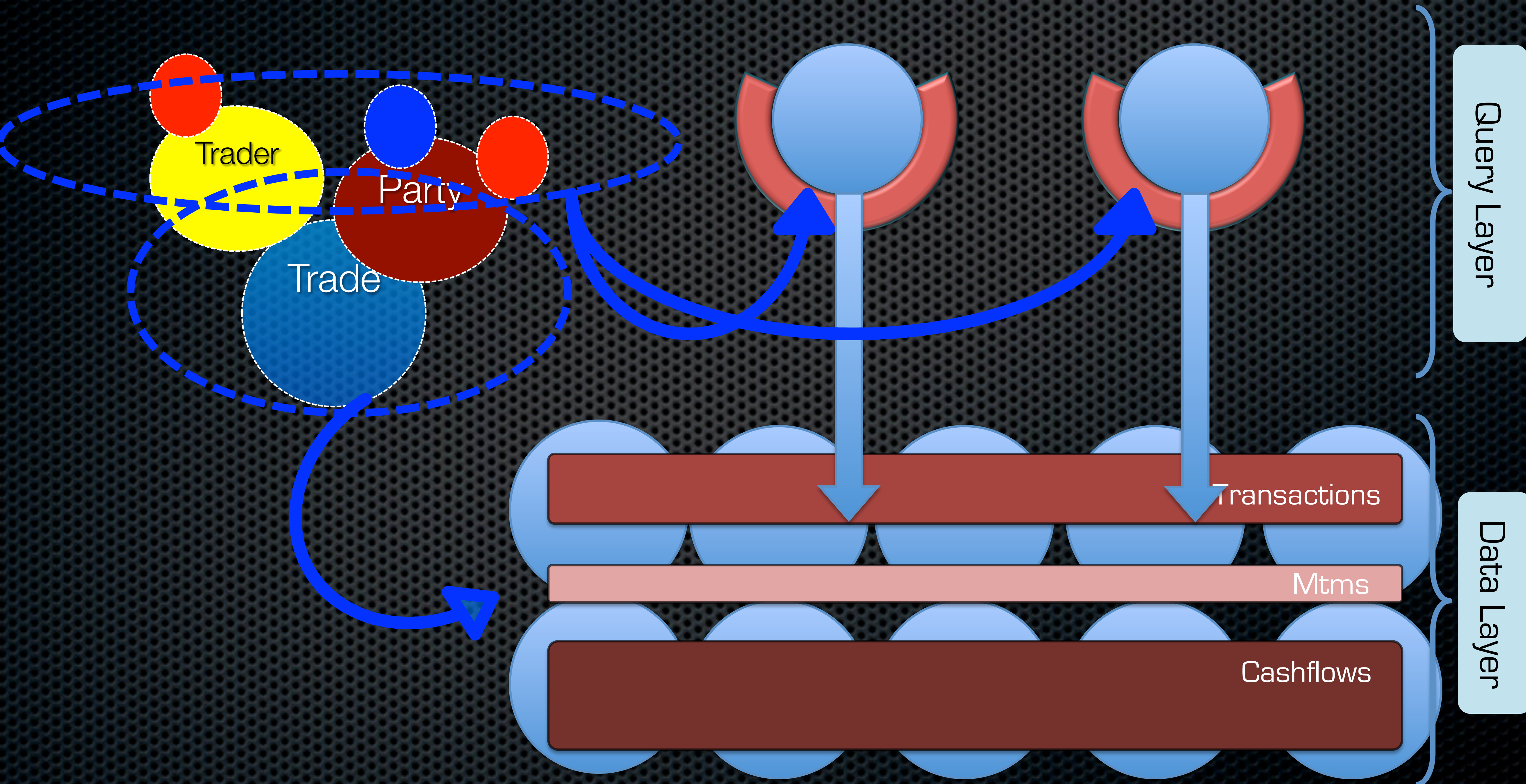
So we prescribe different physical storage for Facts and Dimensions



Facts are held distributed, Dimensions are replicated



- Facts are partitioned across the data layer
- Dimensions are replicated across the Query Layer



Key Point

We use a variant on a Snowflake Schema to partition big stuff, that has the same key and replicate small stuff that has crosscutting keys.

So how does they help us to run queries without distributed joins?

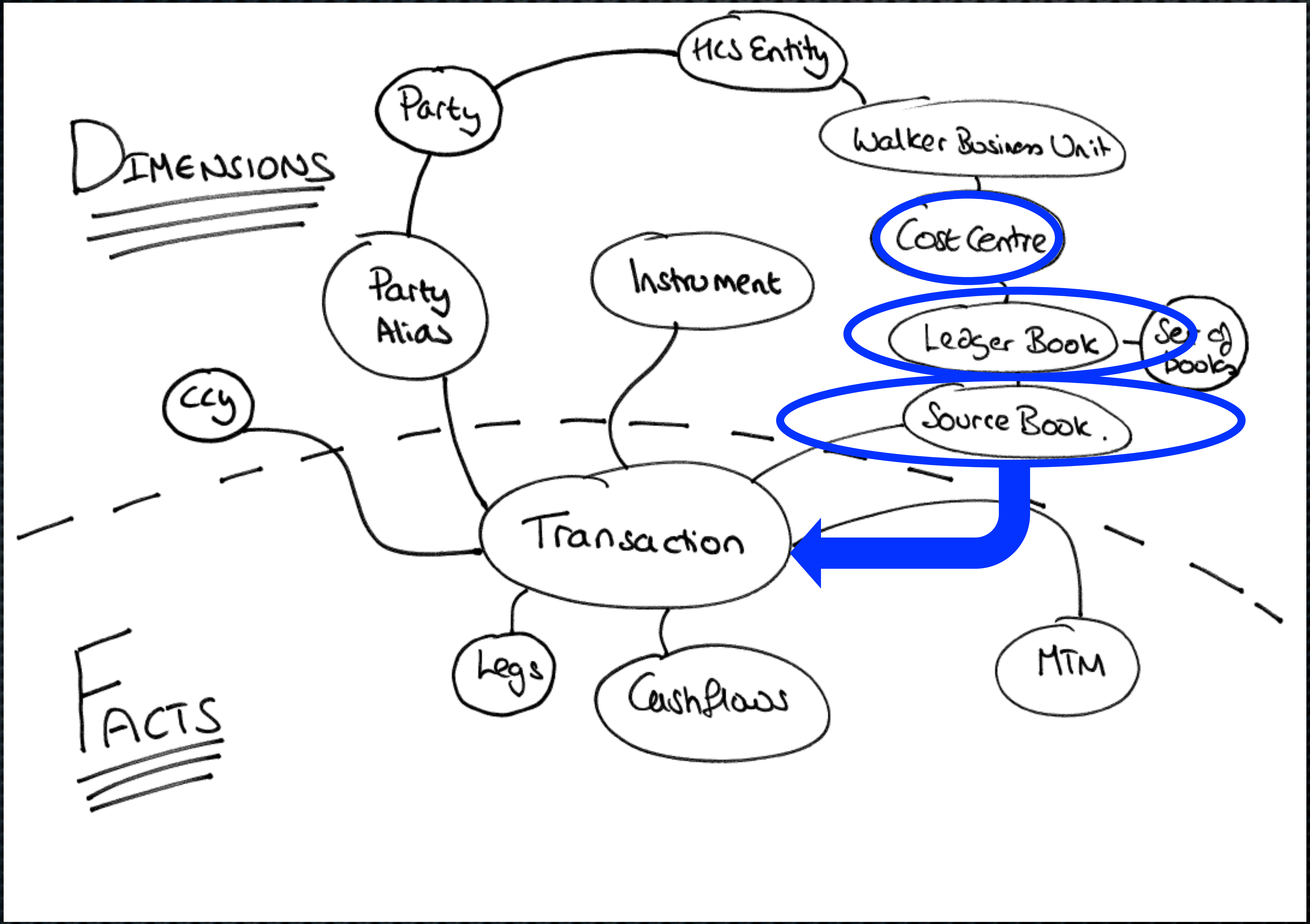
Select Transaction, MTM, ReferenceData From MTM, Transaction, Ref Where Cost Centre = ‘CC1’

This query involves:

- Joins between Dimensions: to evaluate where clause
- Joins between Facts: Transaction joins to MTM
- Joins between all facts and dimensions needed to construct return result

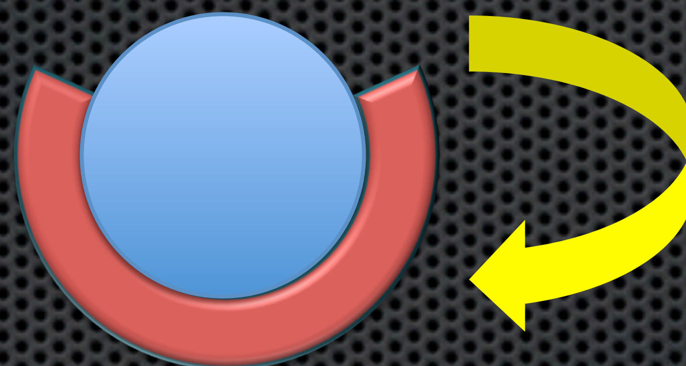
Stage 1: Focus on the where clause:

Where Cost Centre = 'CC1'



Stage 1: Get the right keys to query the Facts

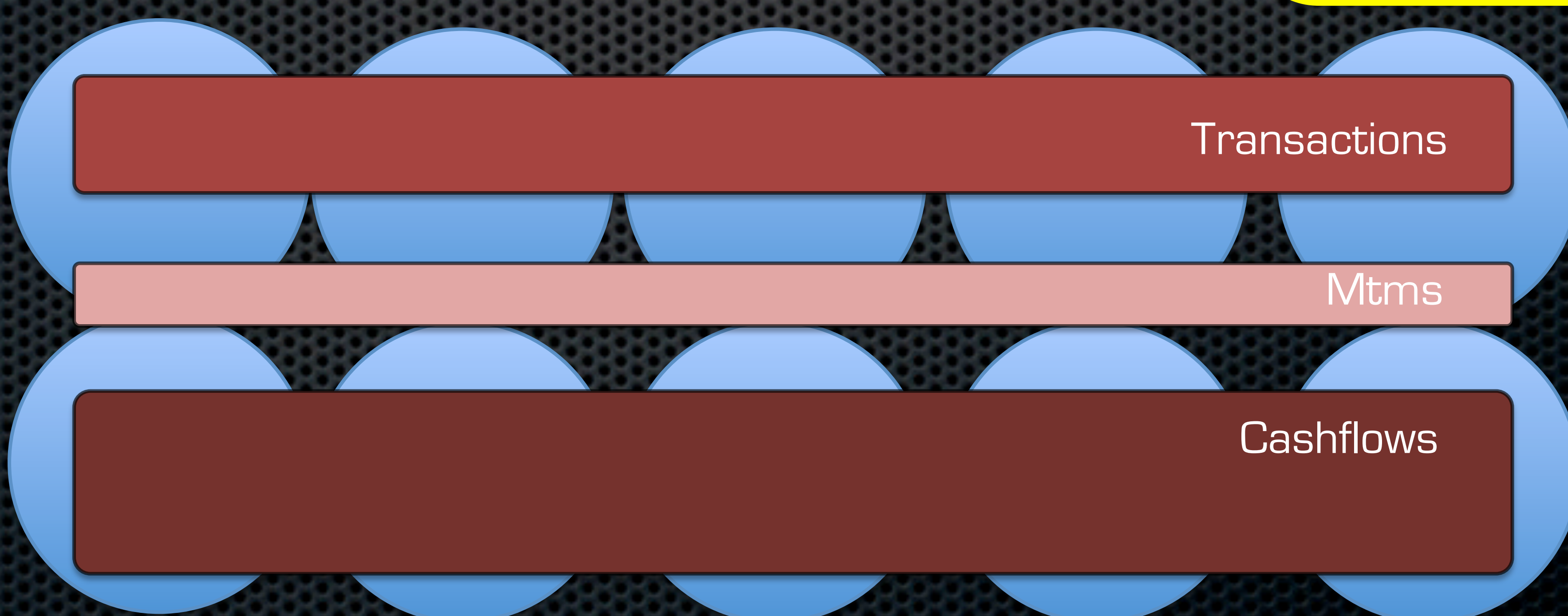
Select Transaction, MTM, ReferenceData From MTM, Transaction, Ref Where Cost Centre = 'CC1'



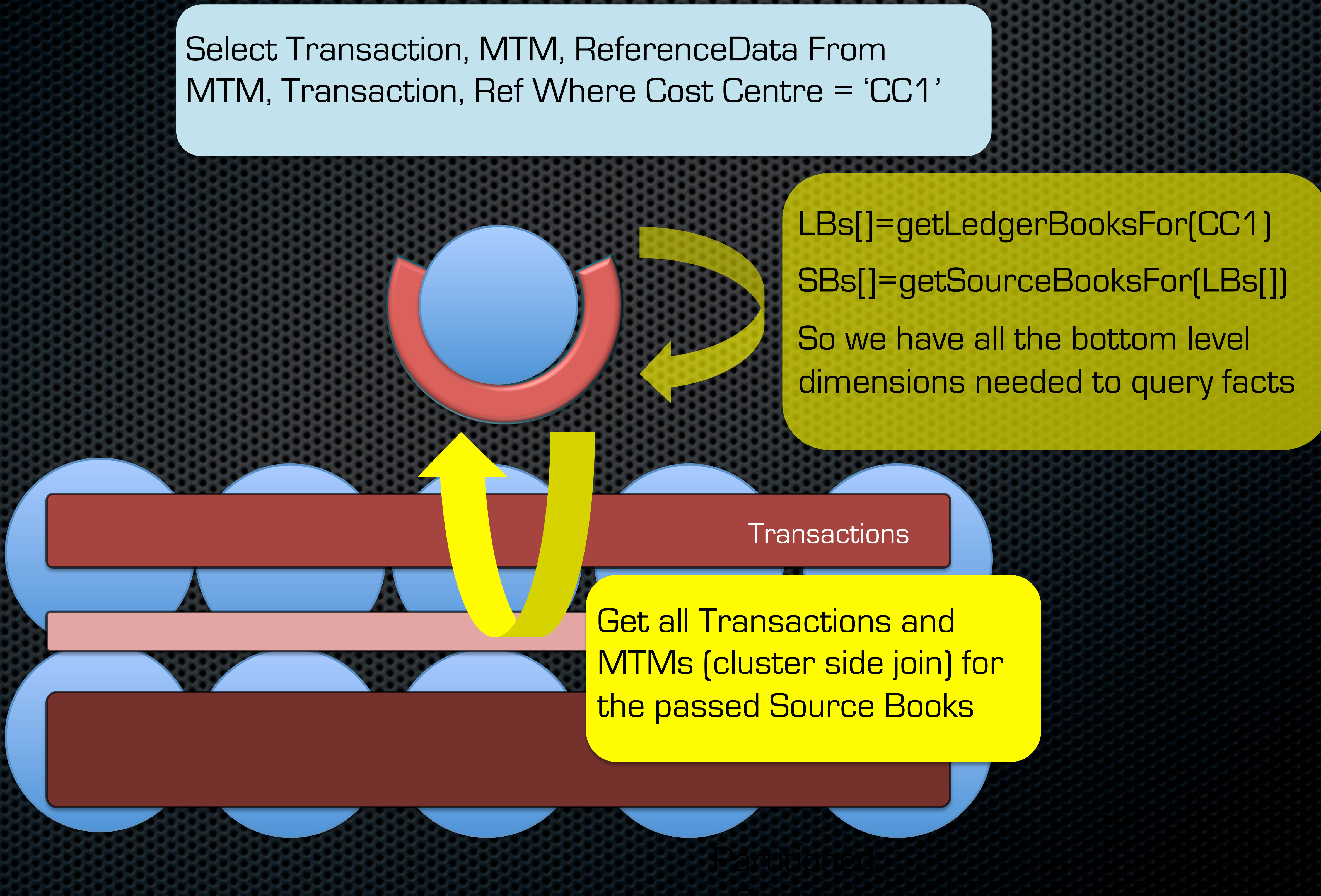
LBs[]=getLedgerBooksFor(CC1)

SBs[]=getSourceBooksFor(LBs[])

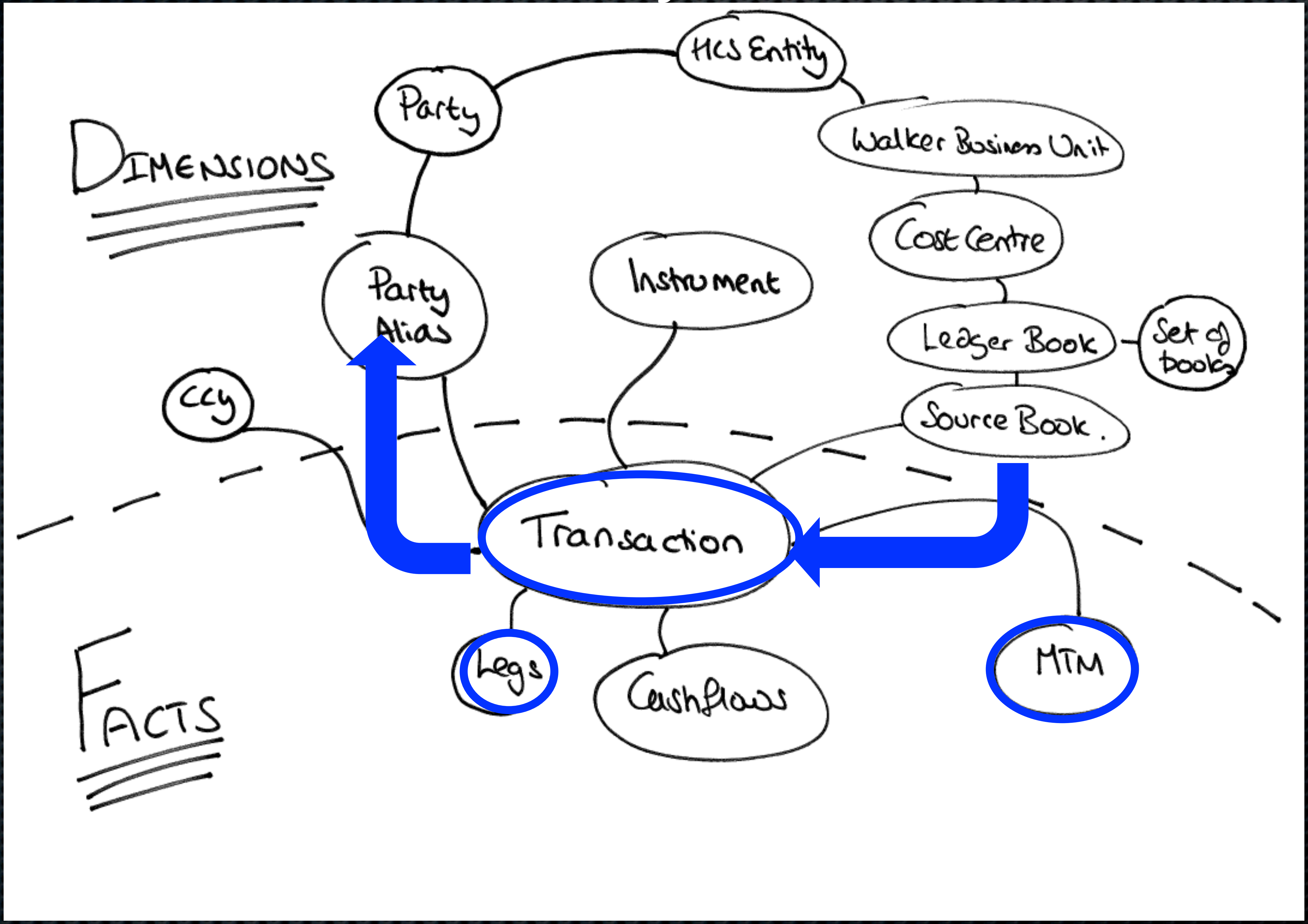
So we have all the bottom level dimensions needed to query facts



Stage 2: Cluster Join to get Facts



Stage 2: Join the facts together efficiently as we know they are collocated



Stage 3: Augment raw Facts with relevant Dimensions

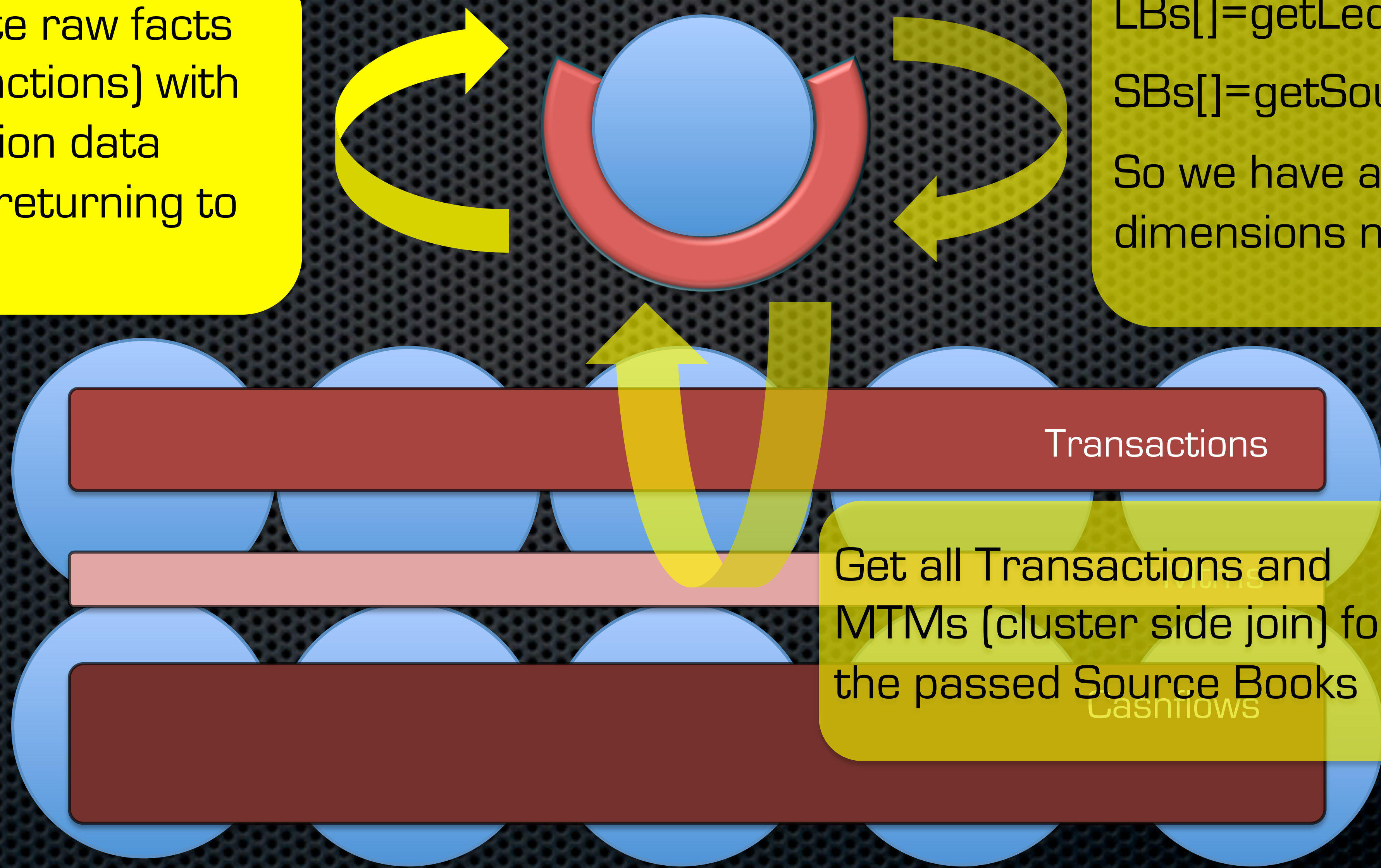
Select Transaction, MTM, ReferenceData From MTM, Transaction, Ref Where Cost Centre = 'CC1'

Populate raw facts (Transactions) with dimension data before returning to client.

LBs[]=getLedgerBooksFor(CC1)

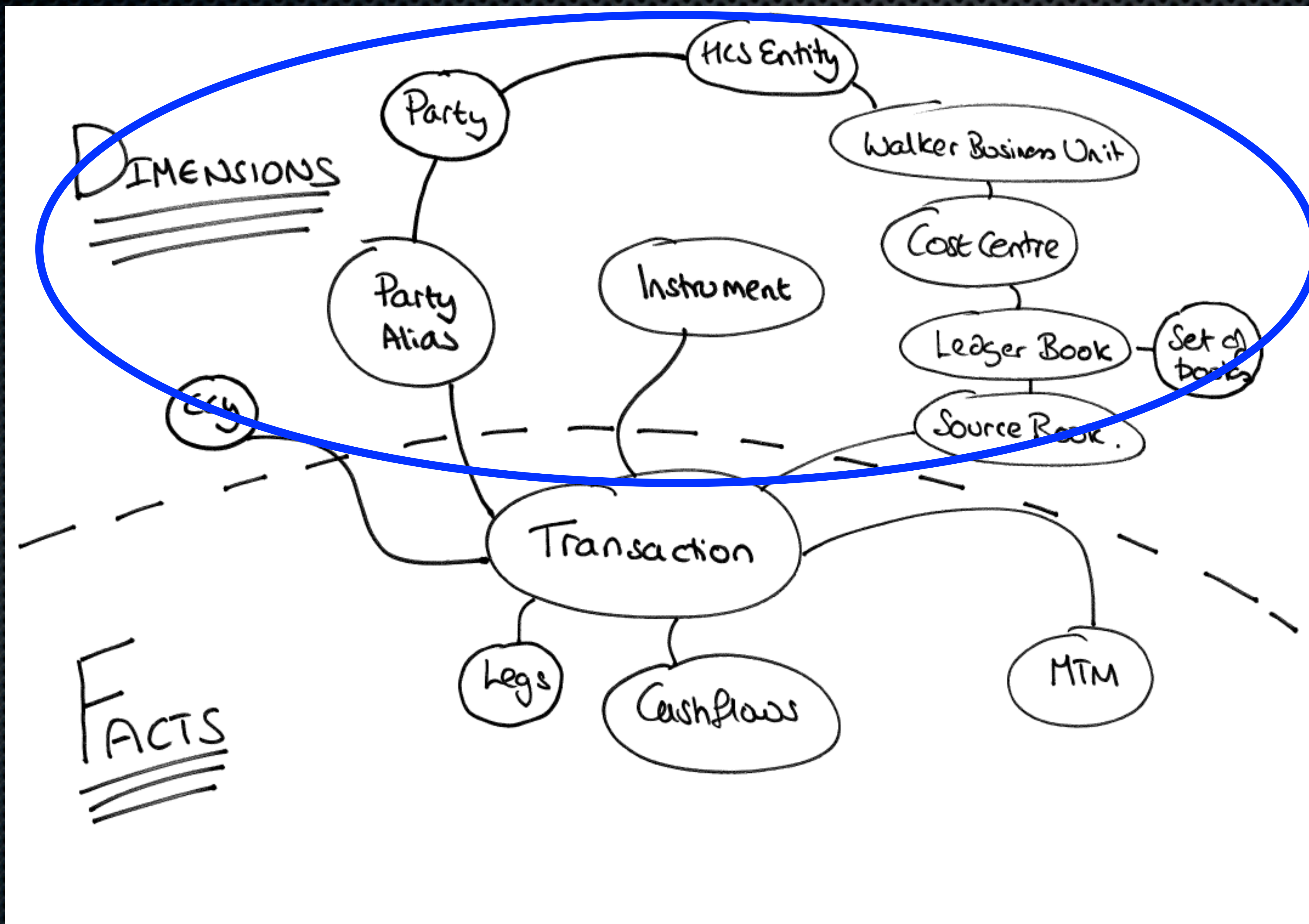
SBs[]=getSourceBooksFor(LBs[])

So we have all the bottom level dimensions needed to query facts

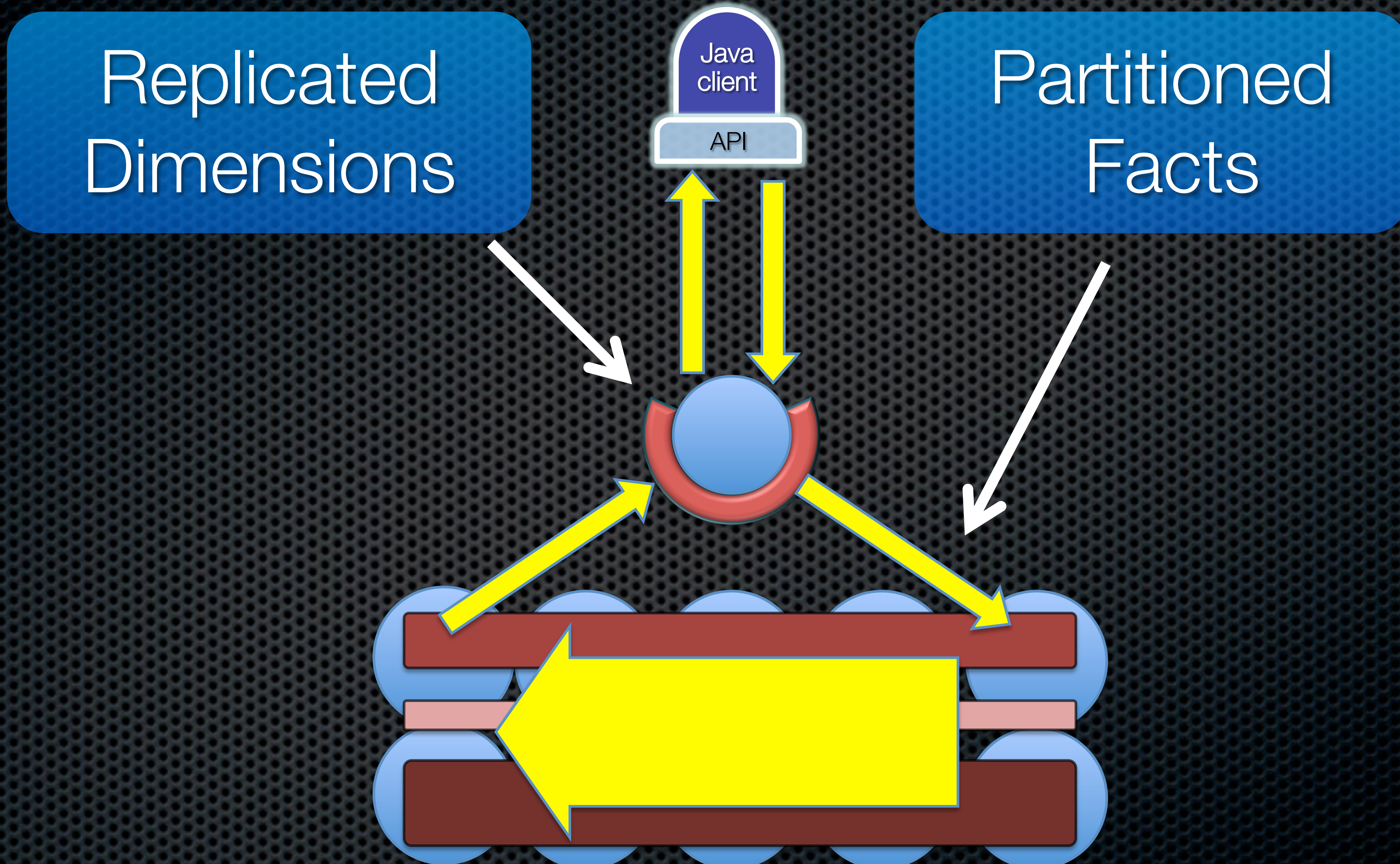


Get all Transactions and MTMs (cluster side join) for the passed Source Books

Stage 3: Bind relevant dimensions to the result



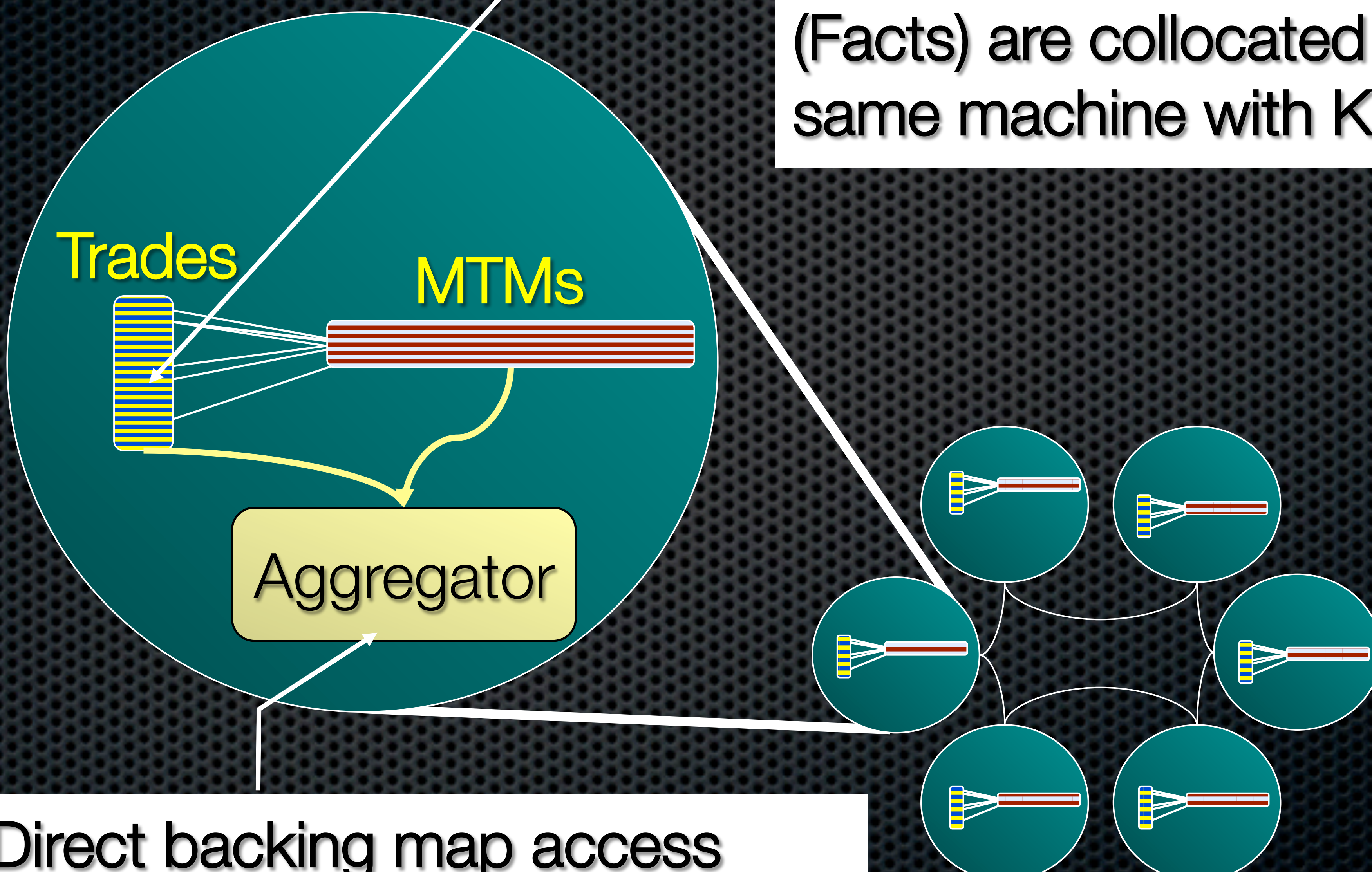
Bringing it together:



We never have to do a distributed join!

Coherence Voodoo: Joining Distributed Facts across the Cluster

Related Trades and MTMs (Facts) are collocated on the same machine with Key Affinity.



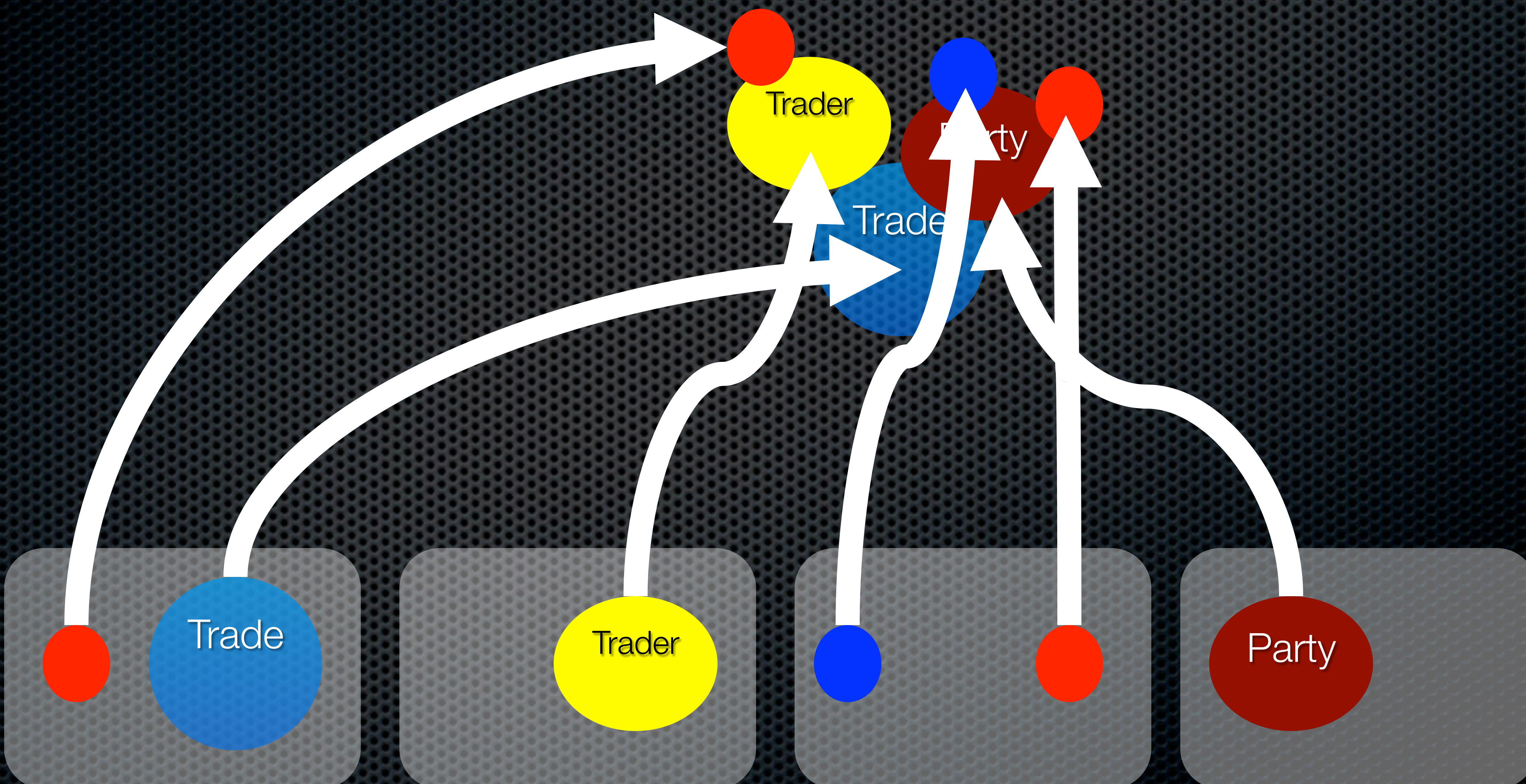
Direct backing map access must be used due to threading issues in Coherence

http://www.benstopford.com/2009/11/20/how-to-perform-efficient-cross-cache-joins-in-coherence/

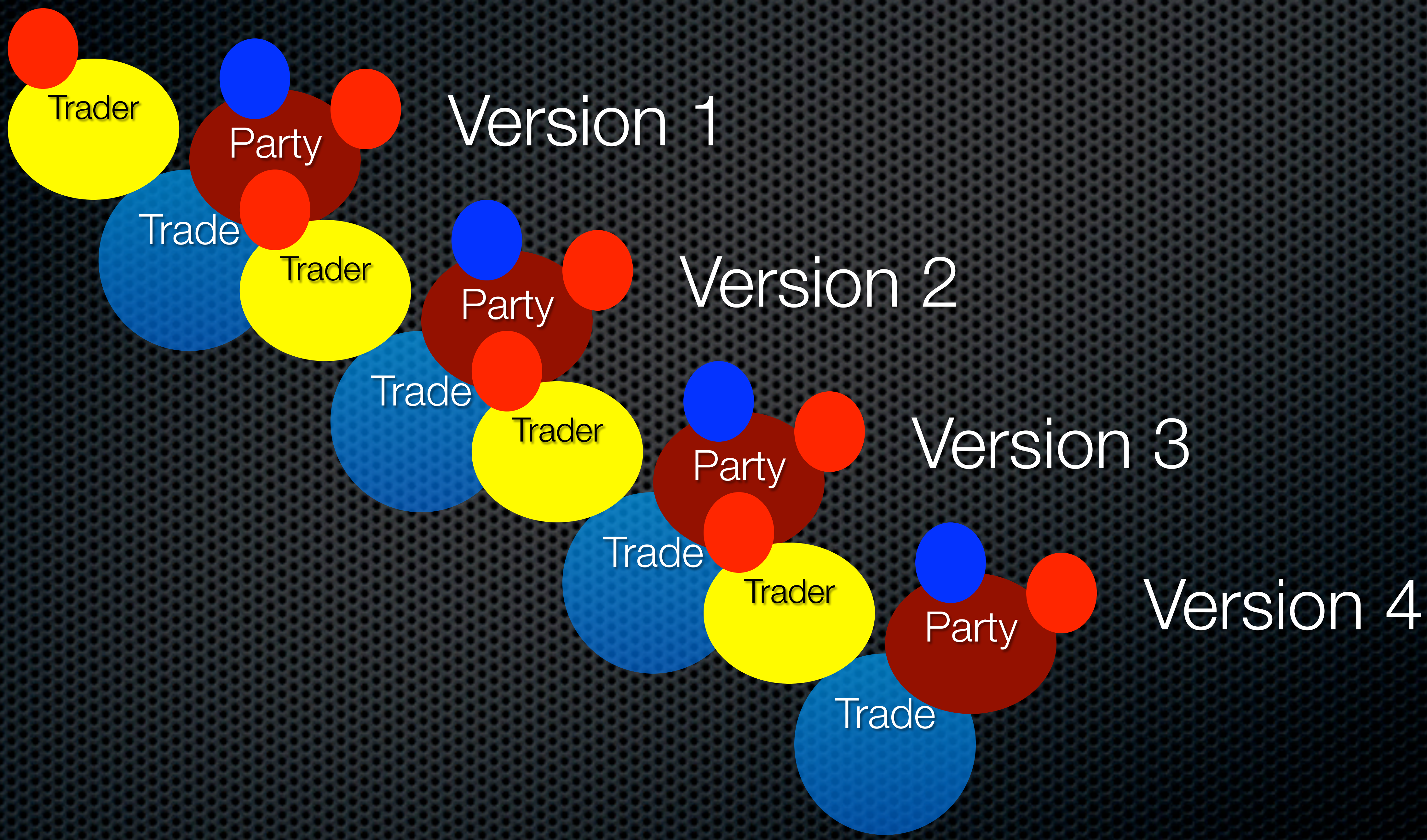
So we are normalised

And we can join without extra network hops

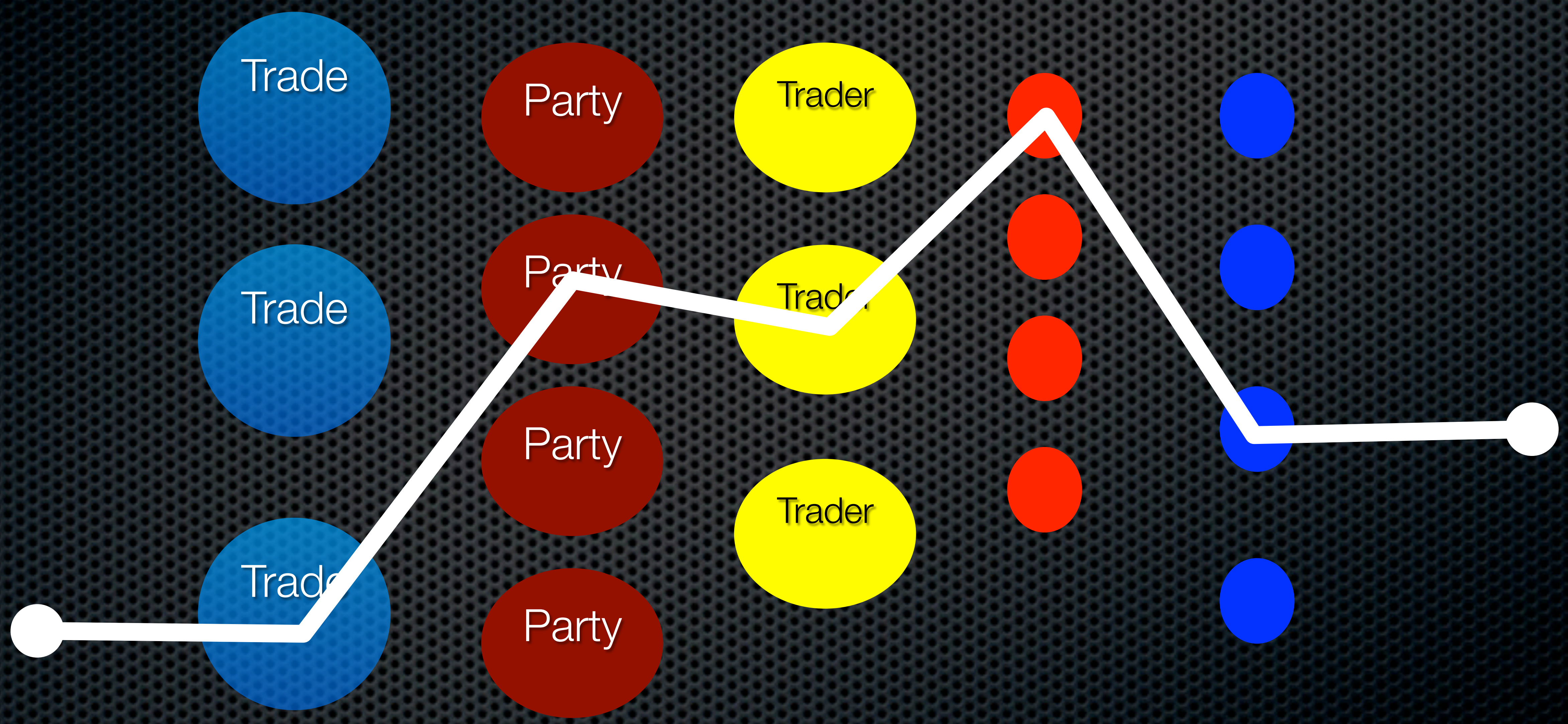
We get to do this...



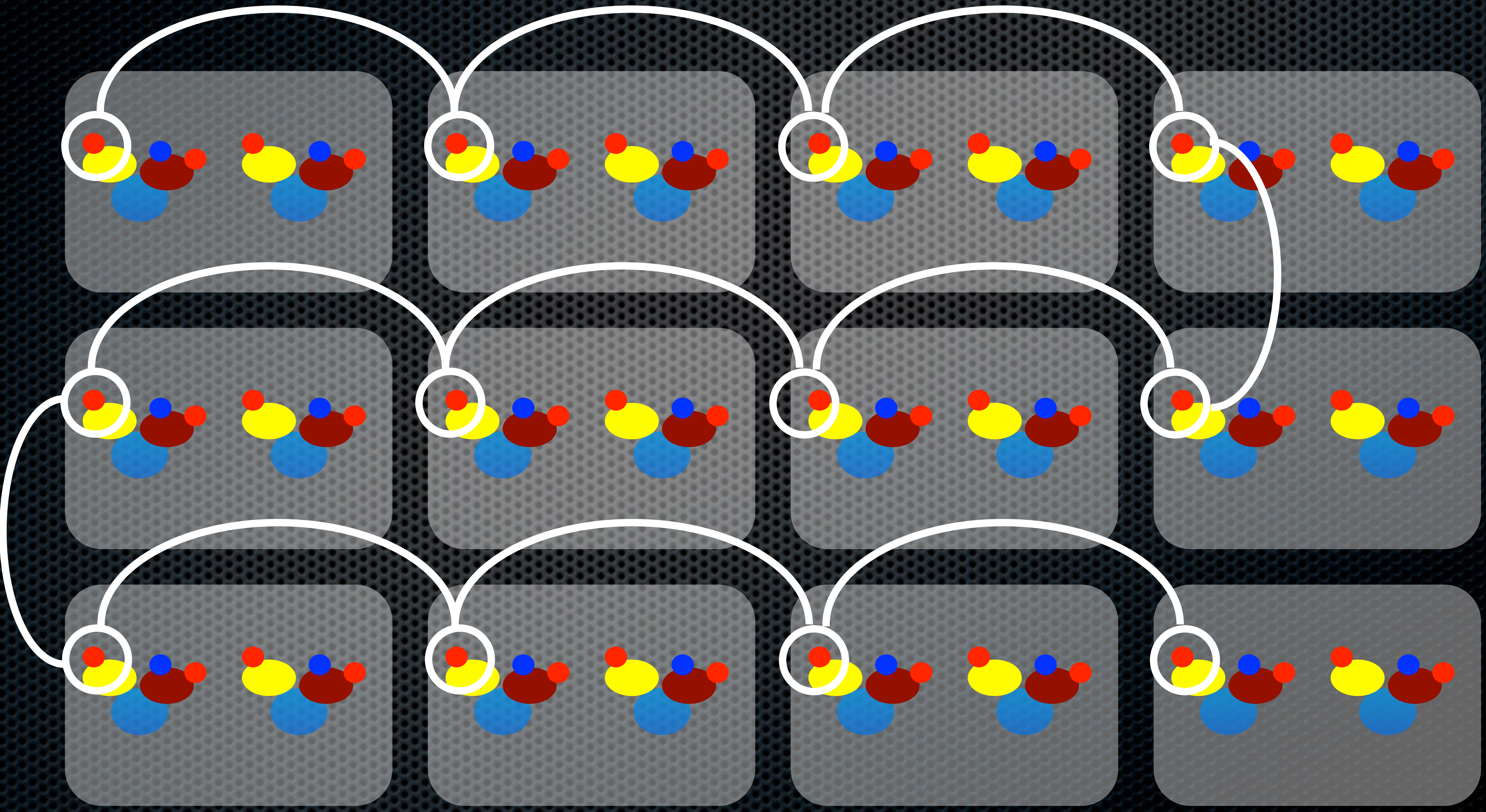
...and this...


Trader
Party
Version 1
Trade
Trader
Party
Version 2
Trade
Trader
Party
Version 3
Trade
Trader
Party
Version 4
Trade

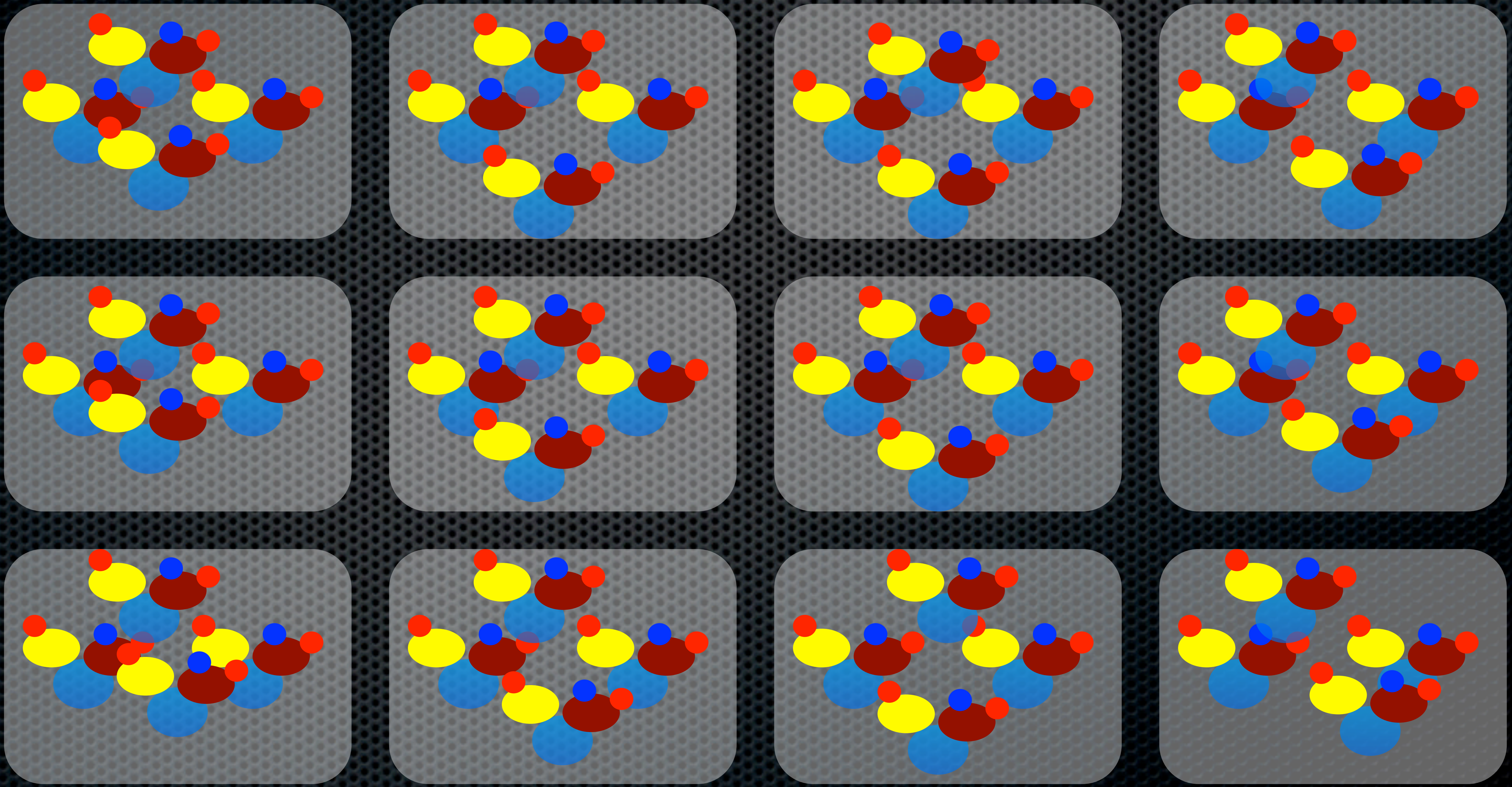
..and this..


Trade
Party
Trader
Trade
Trade
Party
Trader
Trade
Party
Trader
Party

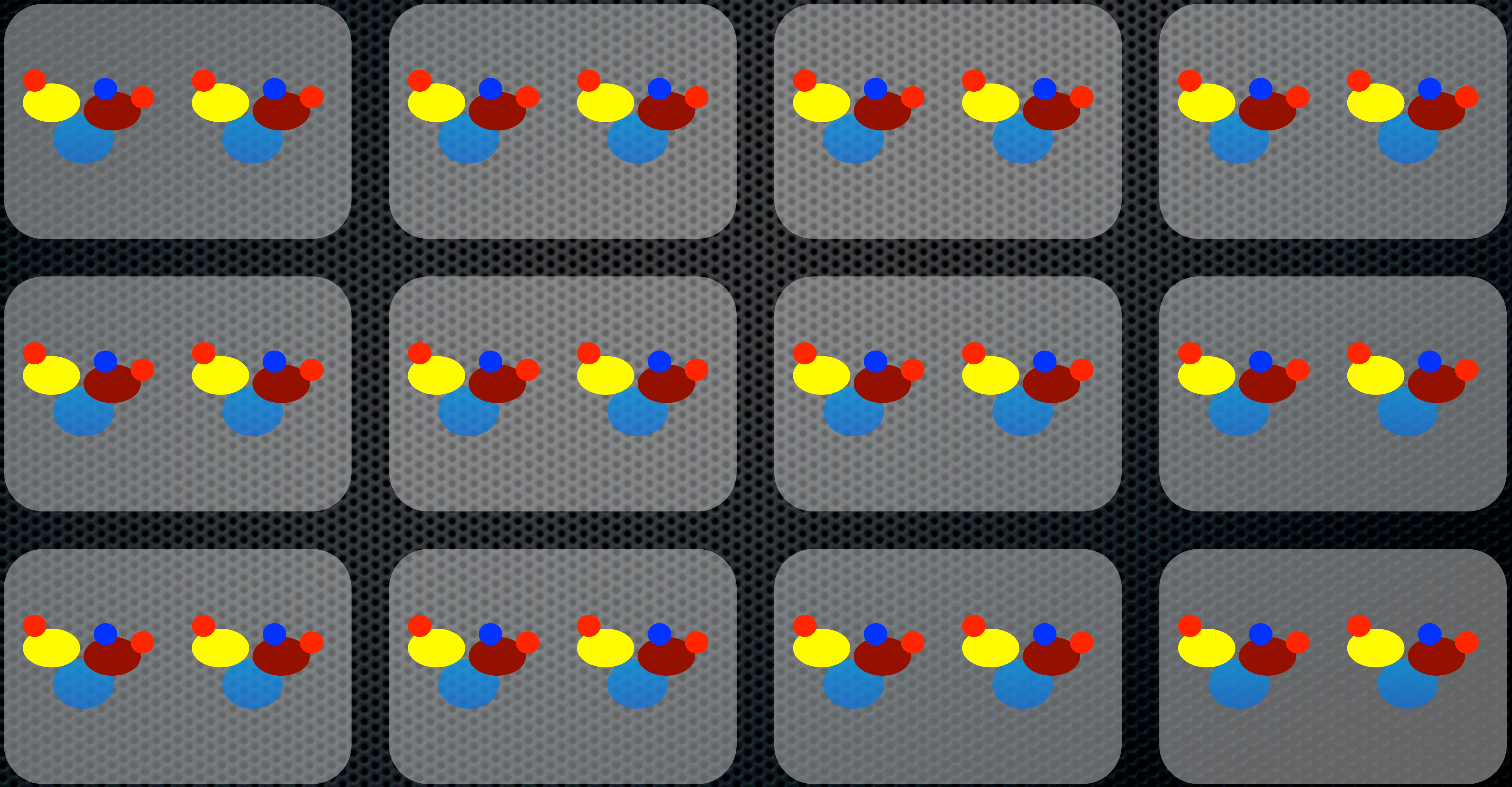
...without the problems of this...



…or this..

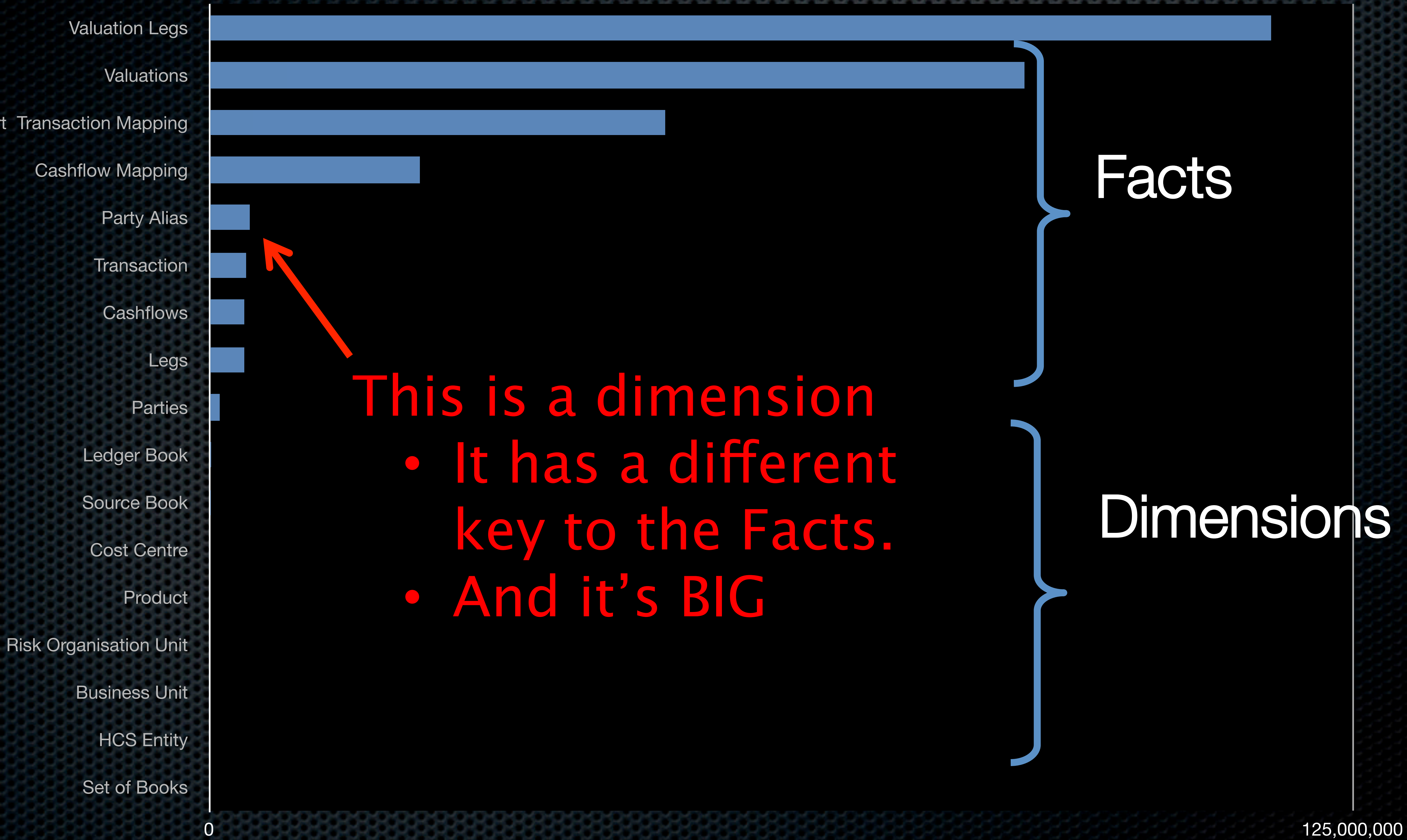


..all at the speed of this... well almost!



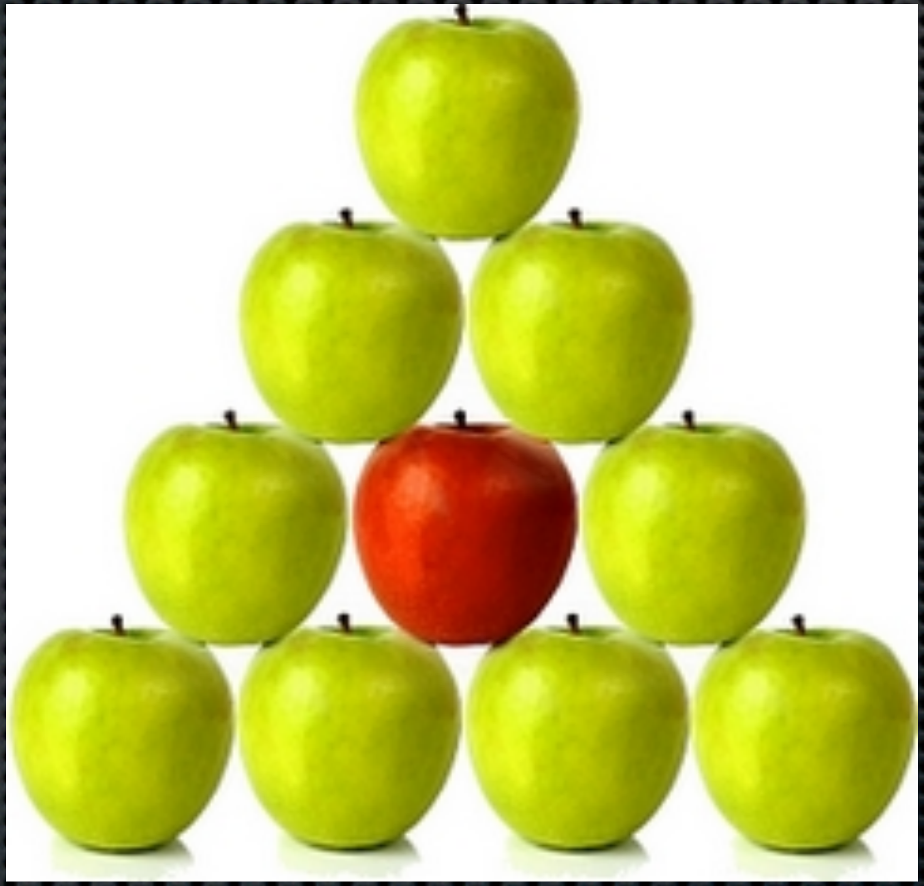
But there is a fly in the ointment…

I lied earlier. These aren't all Facts.



We can’t replicate really big stuff… we’ll run out of space

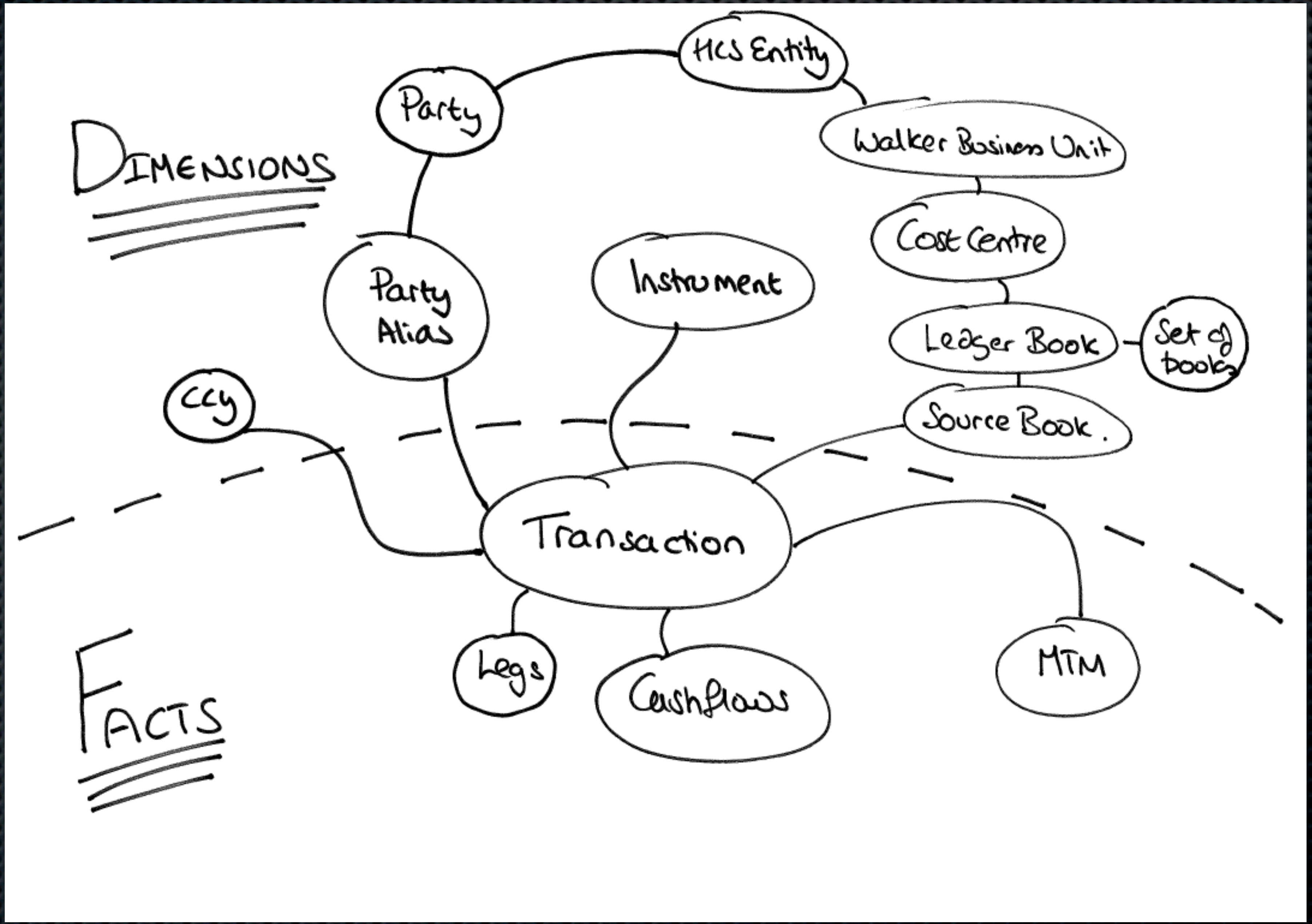
=> Big Dimensions are a problem.



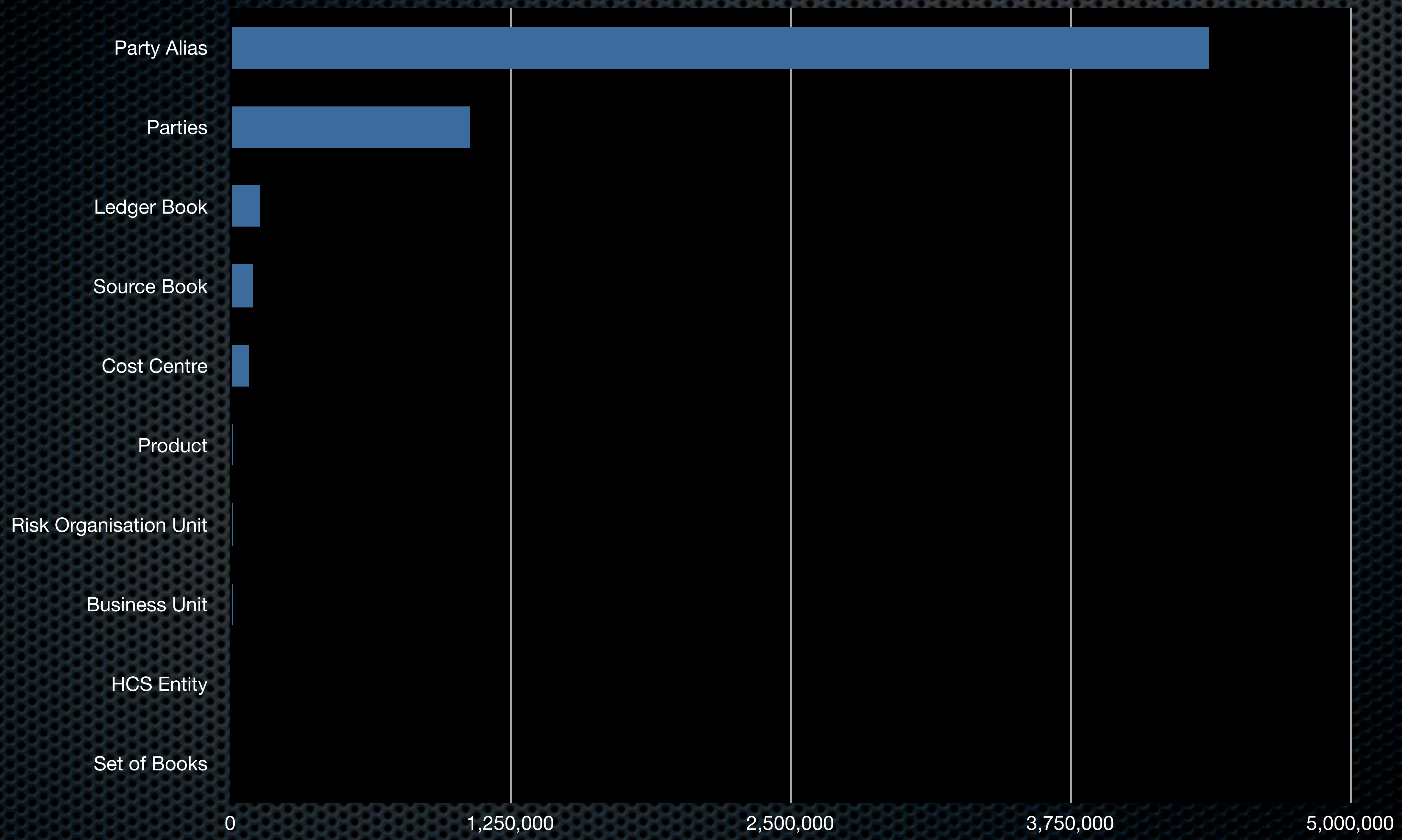
Fortunately we found a simple solution!

We noticed that whilst there are lots of these big dimensions, we didn't actually use a lot of them. They are not all *"connected"*.

If there are no Trades for Barclays in the data store then a Trade Query will never need the Barclays Counterparty



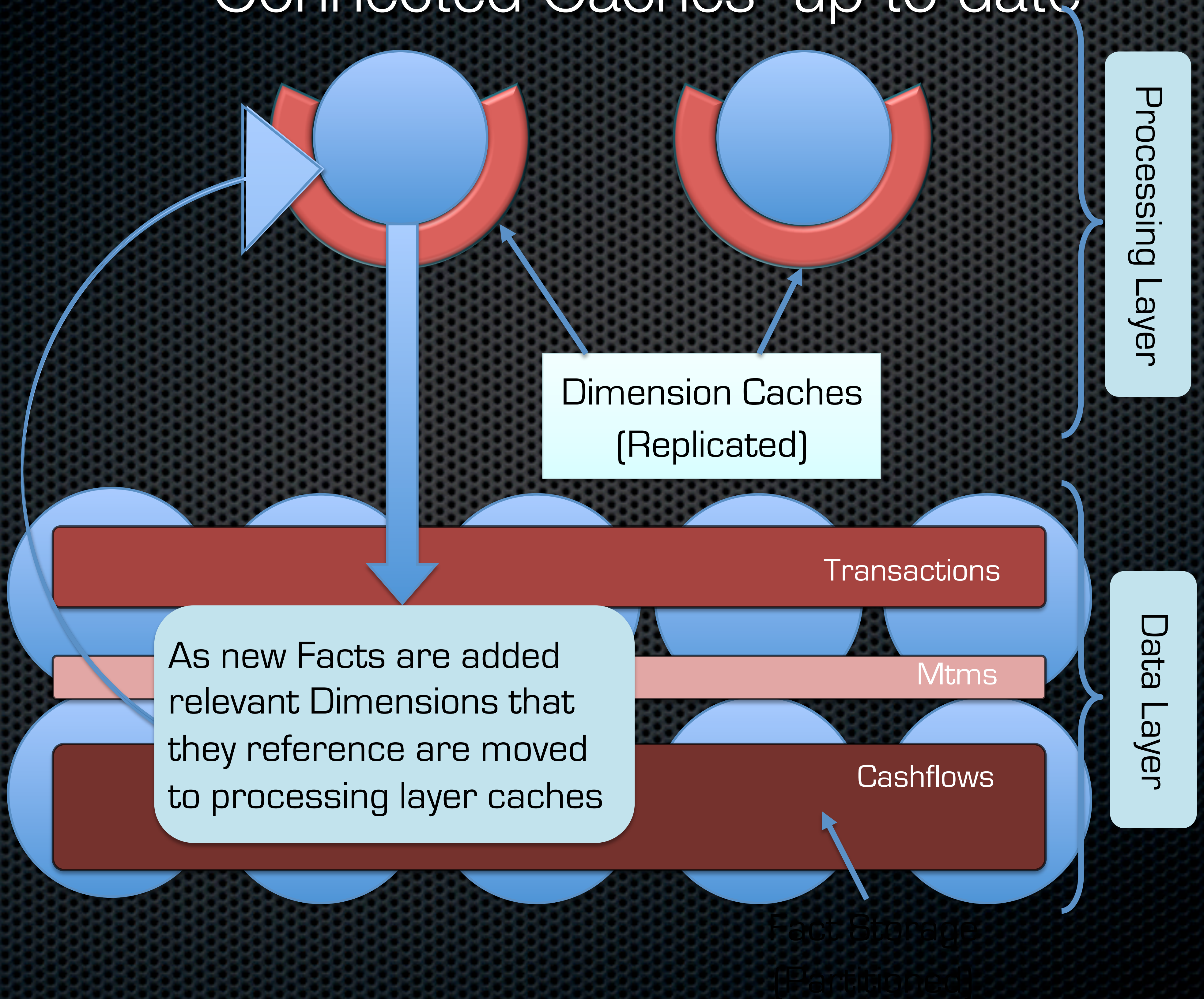
Looking at the All Dimension Data some are quite large



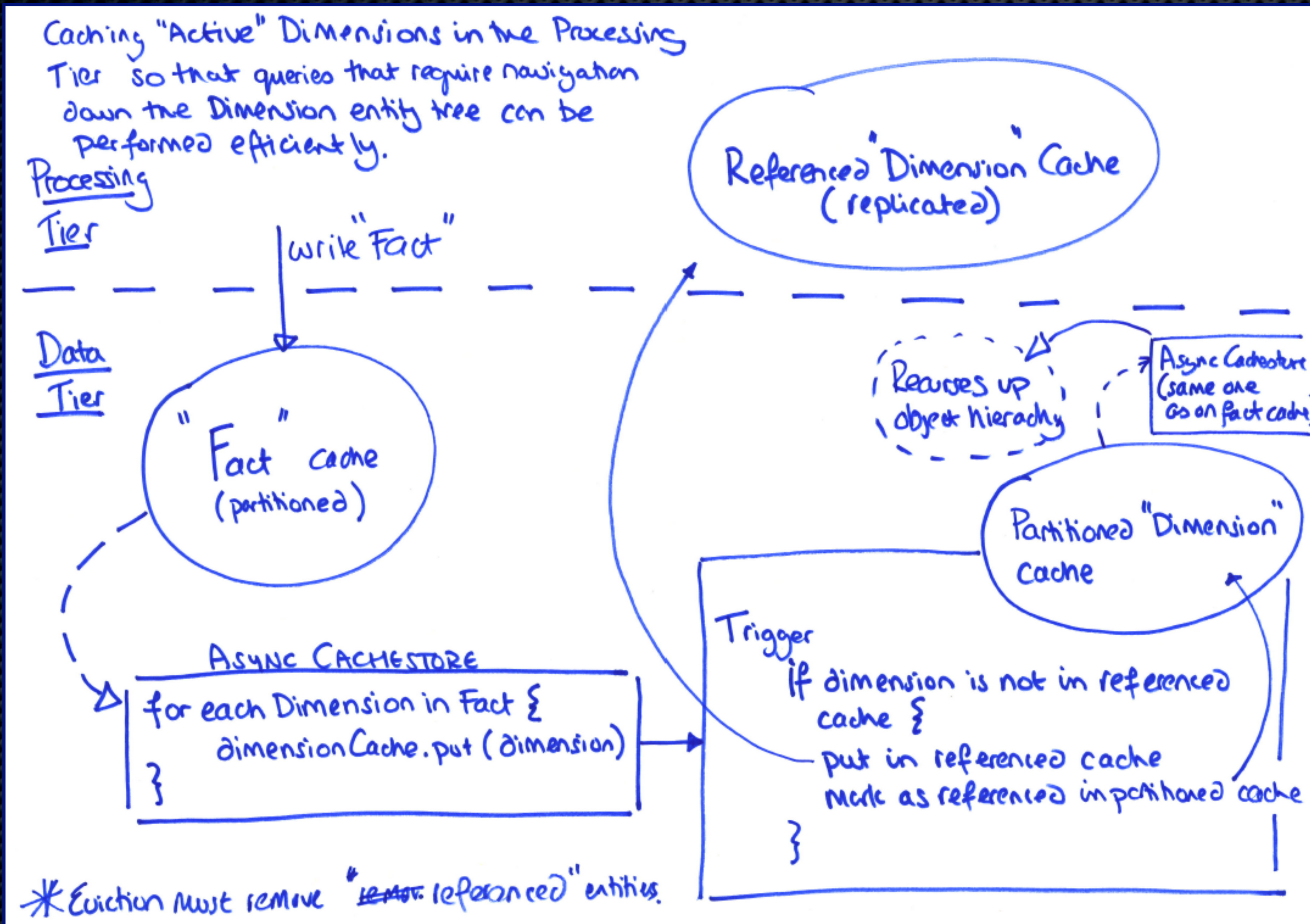
But *Connected* Dimension Data is tiny by comparison

So we only replicate ‘Connected’ or ‘Used’ dimensions

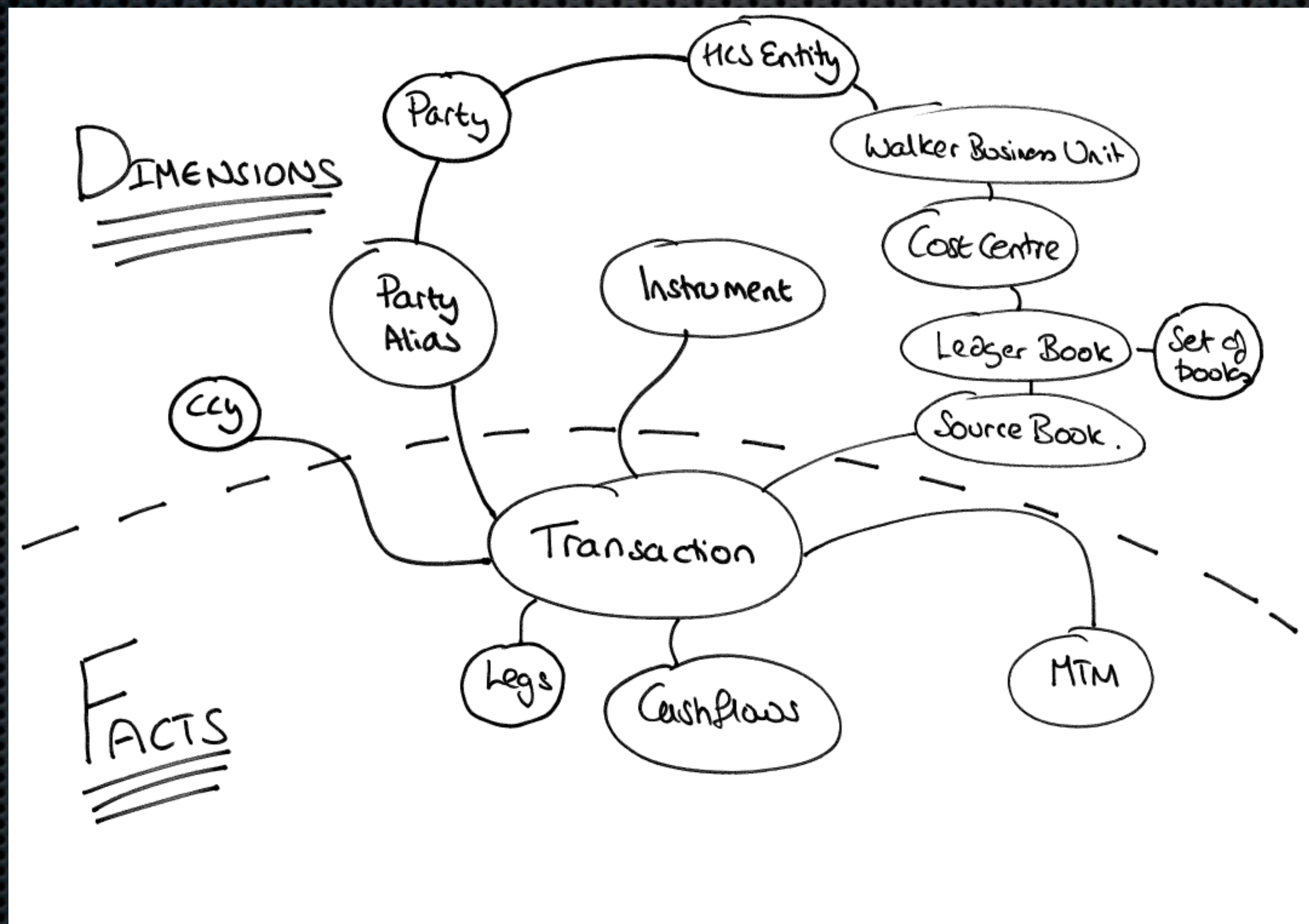
As data is written to the data store we keep our 'Connected Caches' up to date



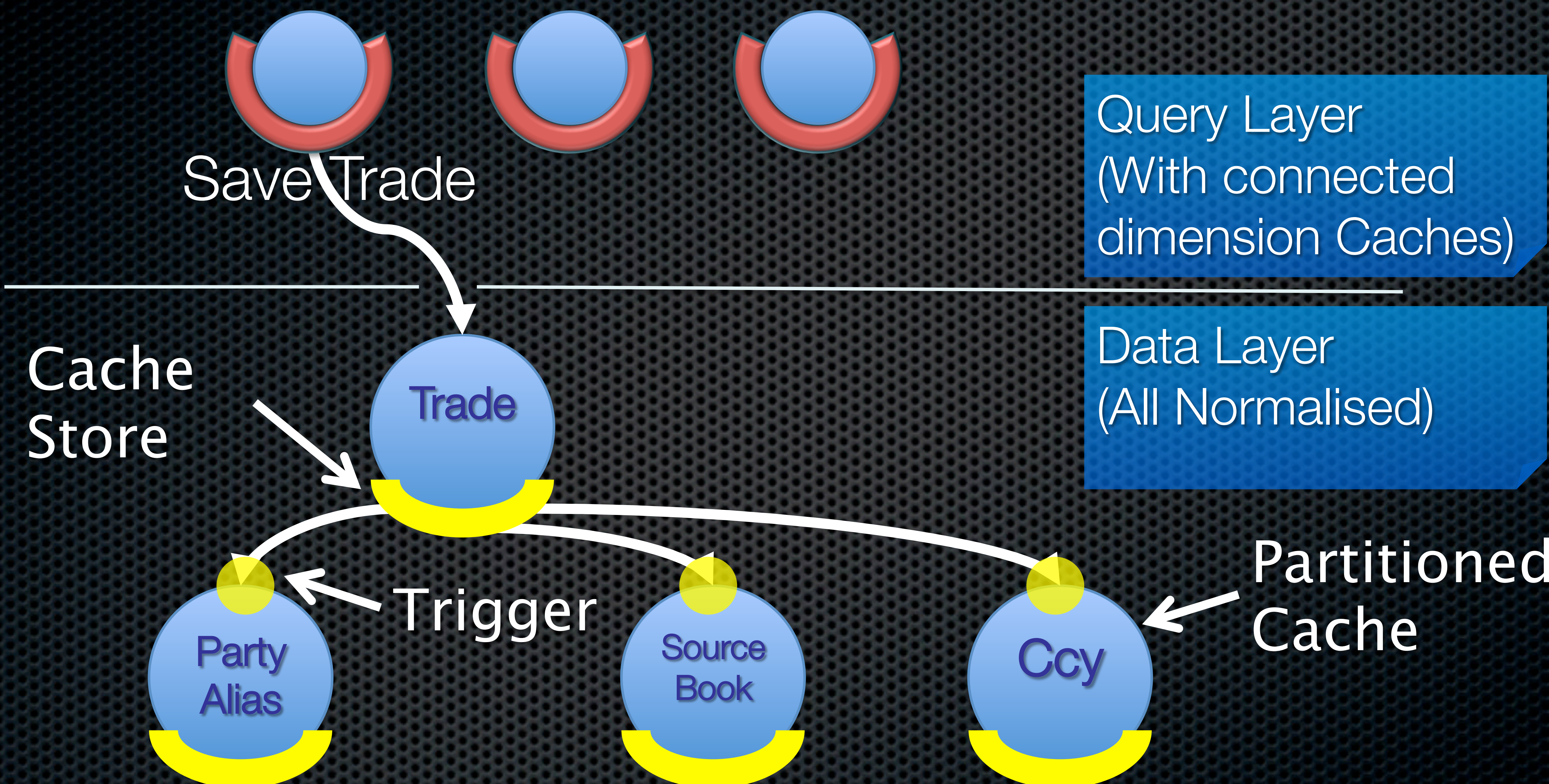
Coherence Voodoo: 'Connected Replication'



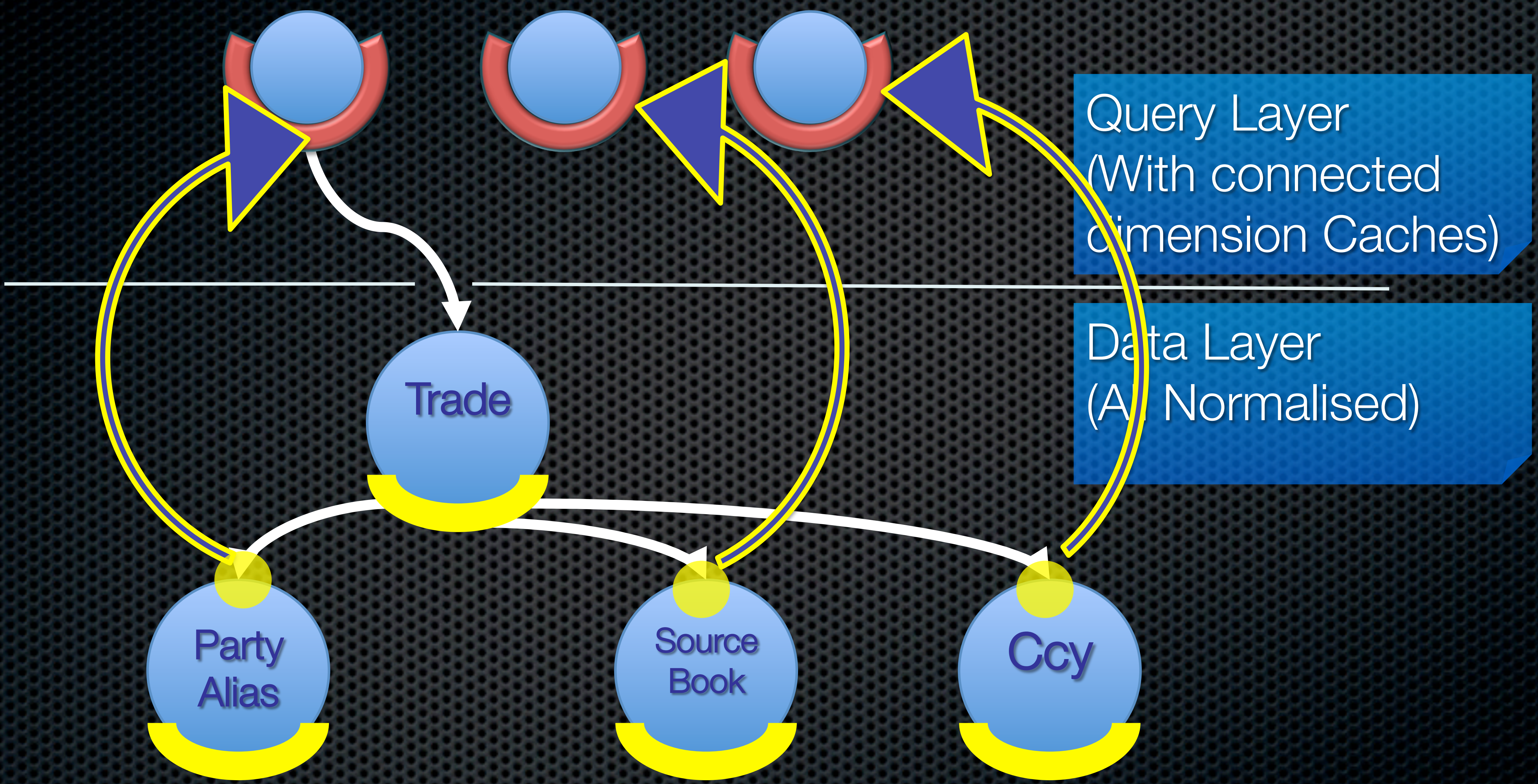
The Replicated Layer is updated by recursing through the arcs on the domain model when facts change



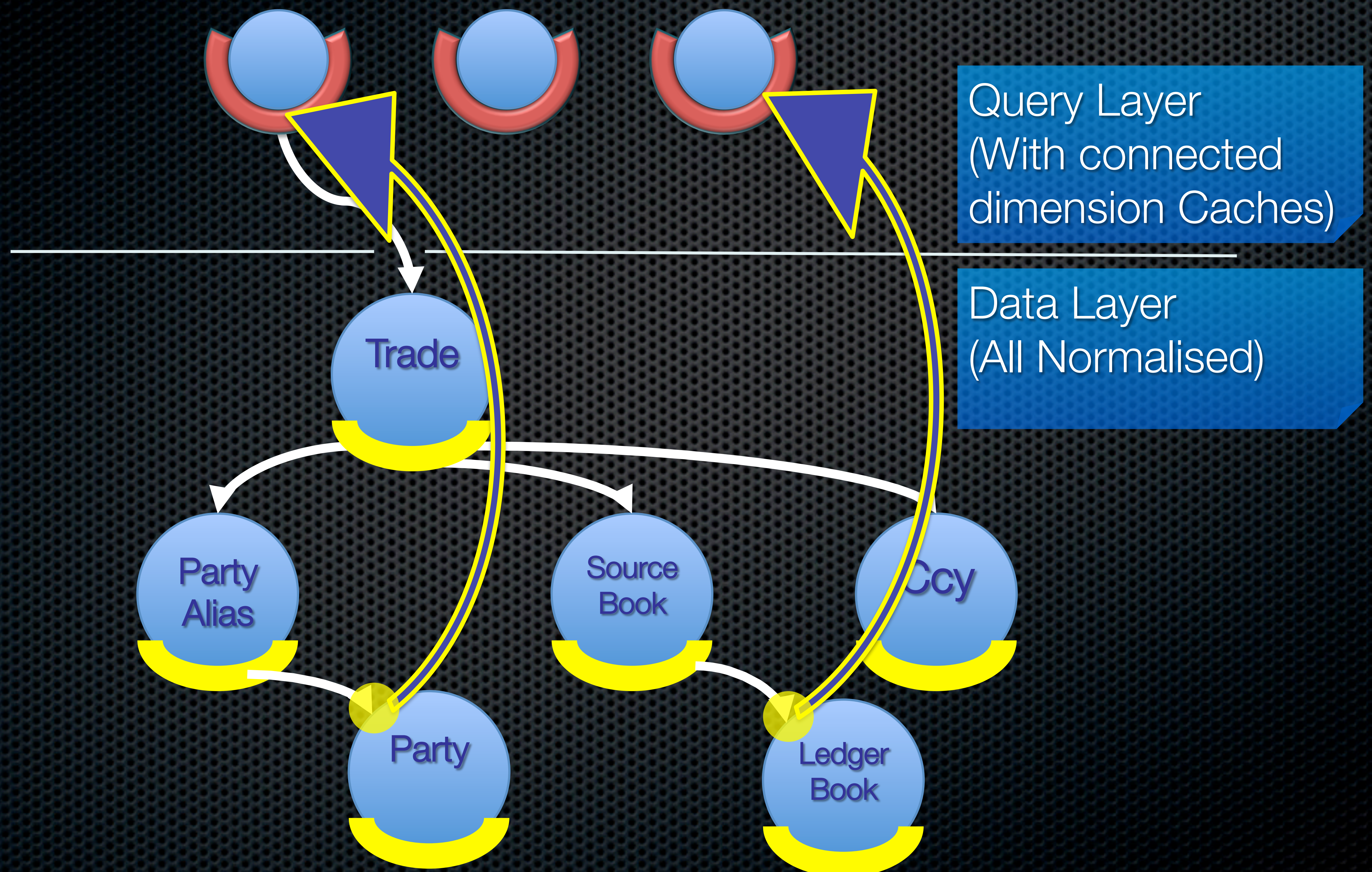
Saving a trade causes all it's 1st level references to be triggered



This updates the connected caches



The process recurses through the object graph



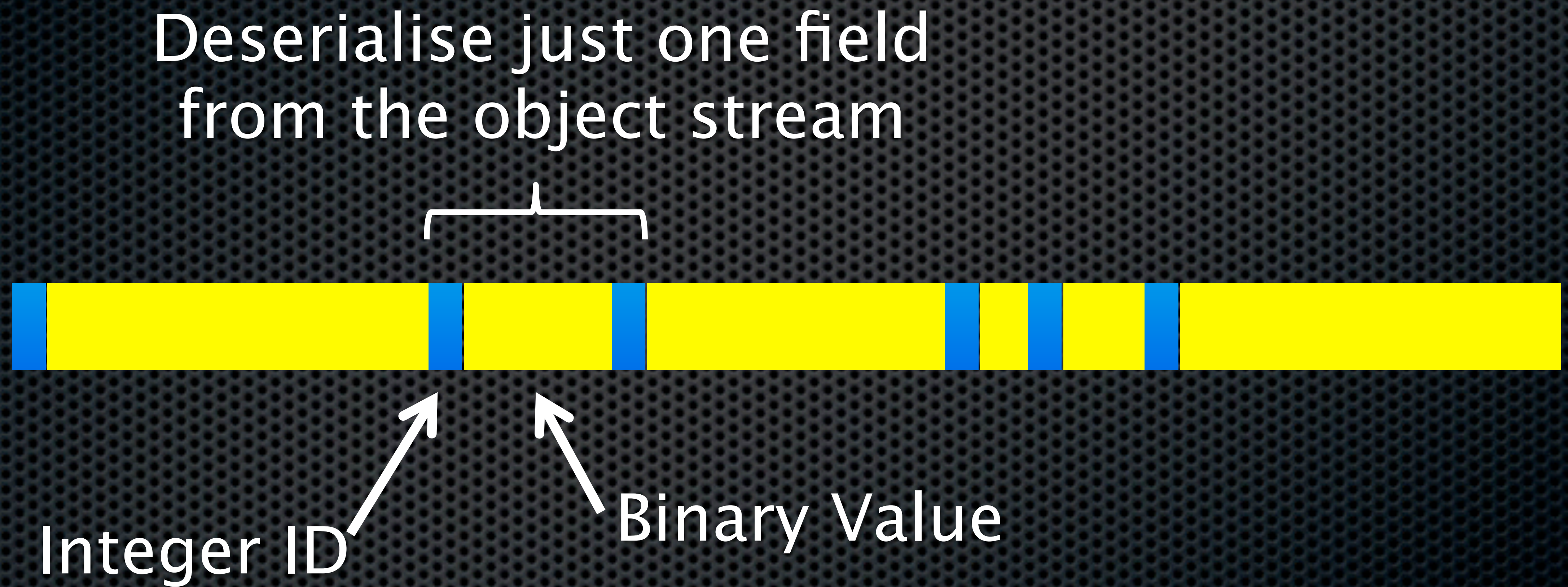
‘Connected Replication’

A simple pattern which recurses through the foreign keys in the domain model, ensuring only ‘Connected’ dimensions are replicated

Limitations of this approach

- Data set size. Size of connected dimensions limits scalability.
- Joins are only supported between *“Facts”* that can share a partitioning key (But any dimension join can be supported)

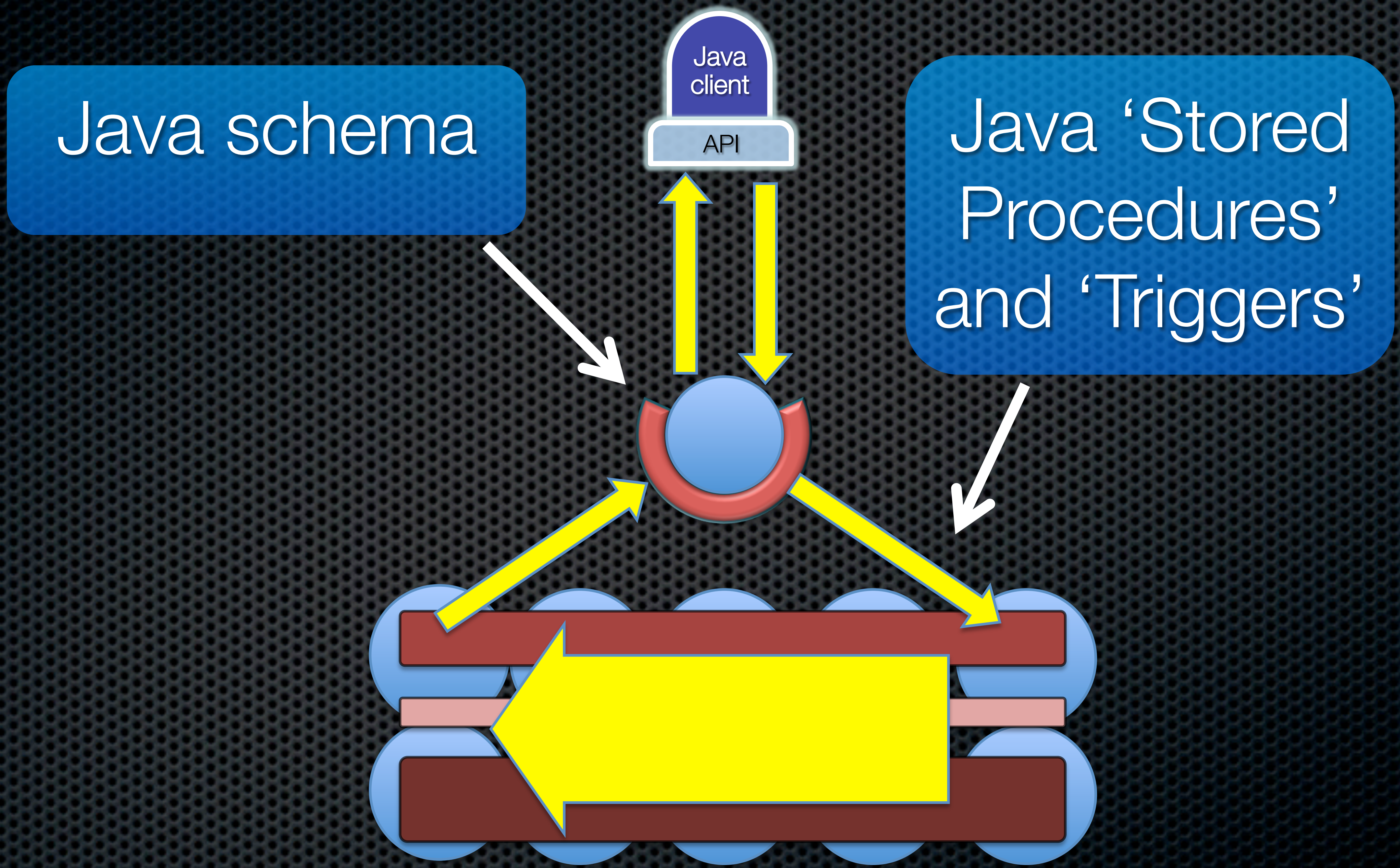
Performance is very sensitive to serialisation costs: Avoid with POF



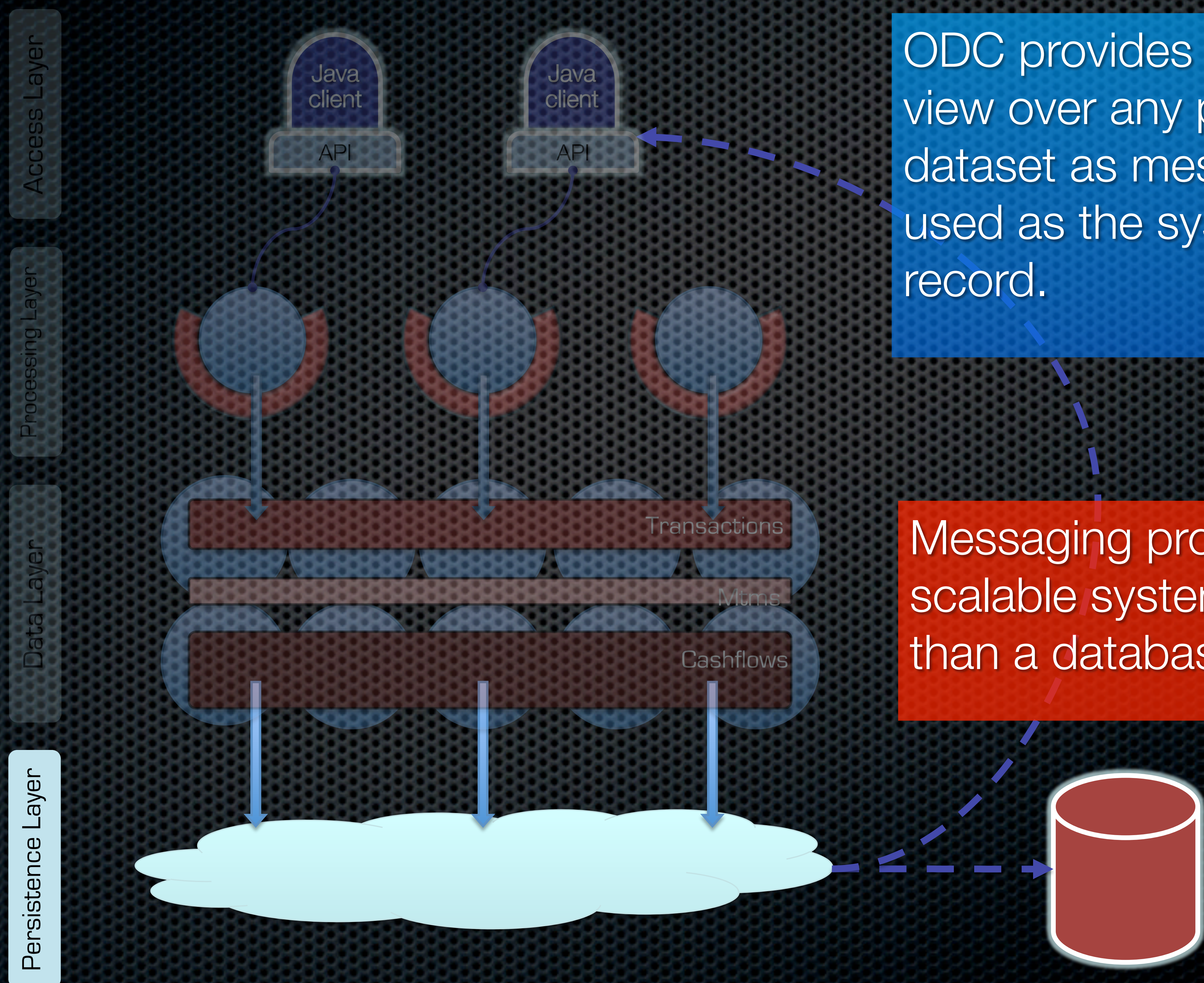
Other cool stuff

(very briefly)

Everything is Java



Messaging as a System of Record



ODC provides a realtime view over any part of the dataset as messaging is the used as the system of record.

Messaging provides a more scalable system of record than a database would.

Being event based changes the programming model.

- The system provides both real time and query based views on the data.
- The two are linked using versioning
- Replication to DR, DB, fact aggregation

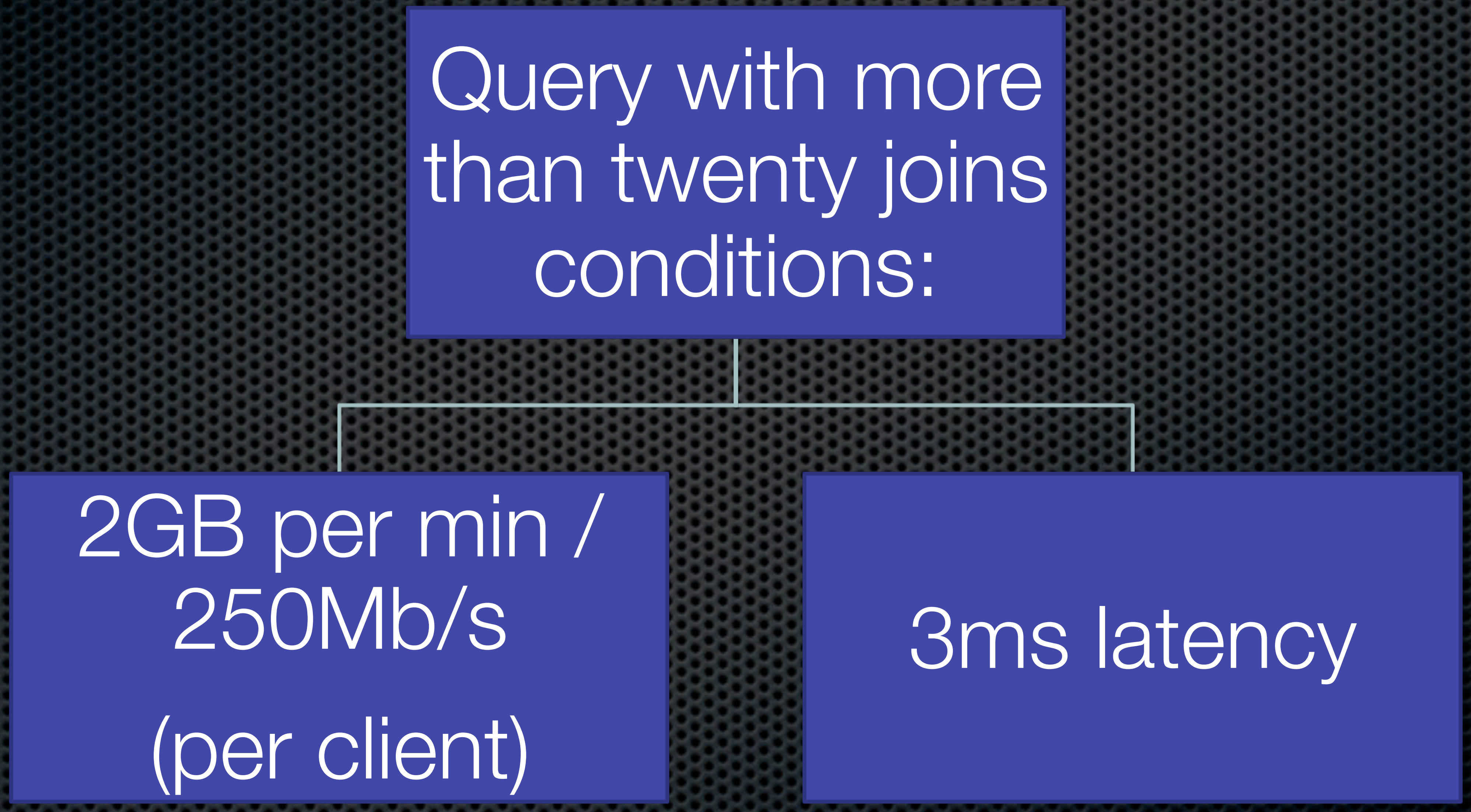
API – Queries utilise a fluent interface

```
public class Sample {
    public Sample() throws ODCAccessException {
        ODCFactory factory = ODCSimpleStart.getODCFactory();
        ODC odc = factory.getAccessLayerInstance("Ben", "pa55word");

        ValuationQueryFromDailyTransaction q = odc.buildQuery()
                            .fromTransactionAndValuation()
                            .forSystemInstanceId(SystemInstanceId.ICE_GBLO)
                            .forBookId()
        ;
        odc.query(q)
    }
```

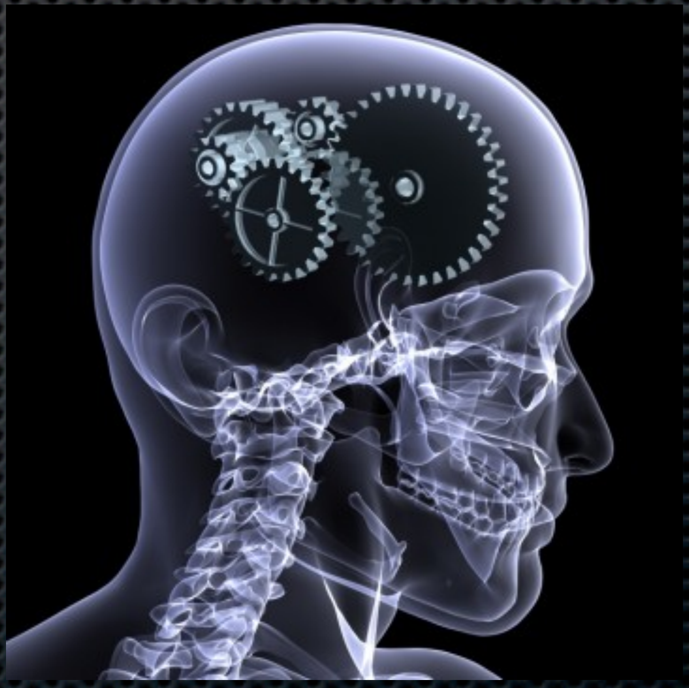
```
forBookId(SourceBookId bookId)                 Valuation(
forVersionedTransactionIds(Set<Va..            Valuation(
forBookIds(Collection<SourceBookI..            Valuation(
forBusinessDate(BusinessDate busi..            Valuation(
forCounterpartyAliases(PartyAlias..            Valuation(
forSystemInstanceId(SystemInstanc..            Valuation(
forTransactionIds(Set<Transaction..            Valuation(
Choosing item with Tab will overwrite the rest of identifier after caret
```

Performance



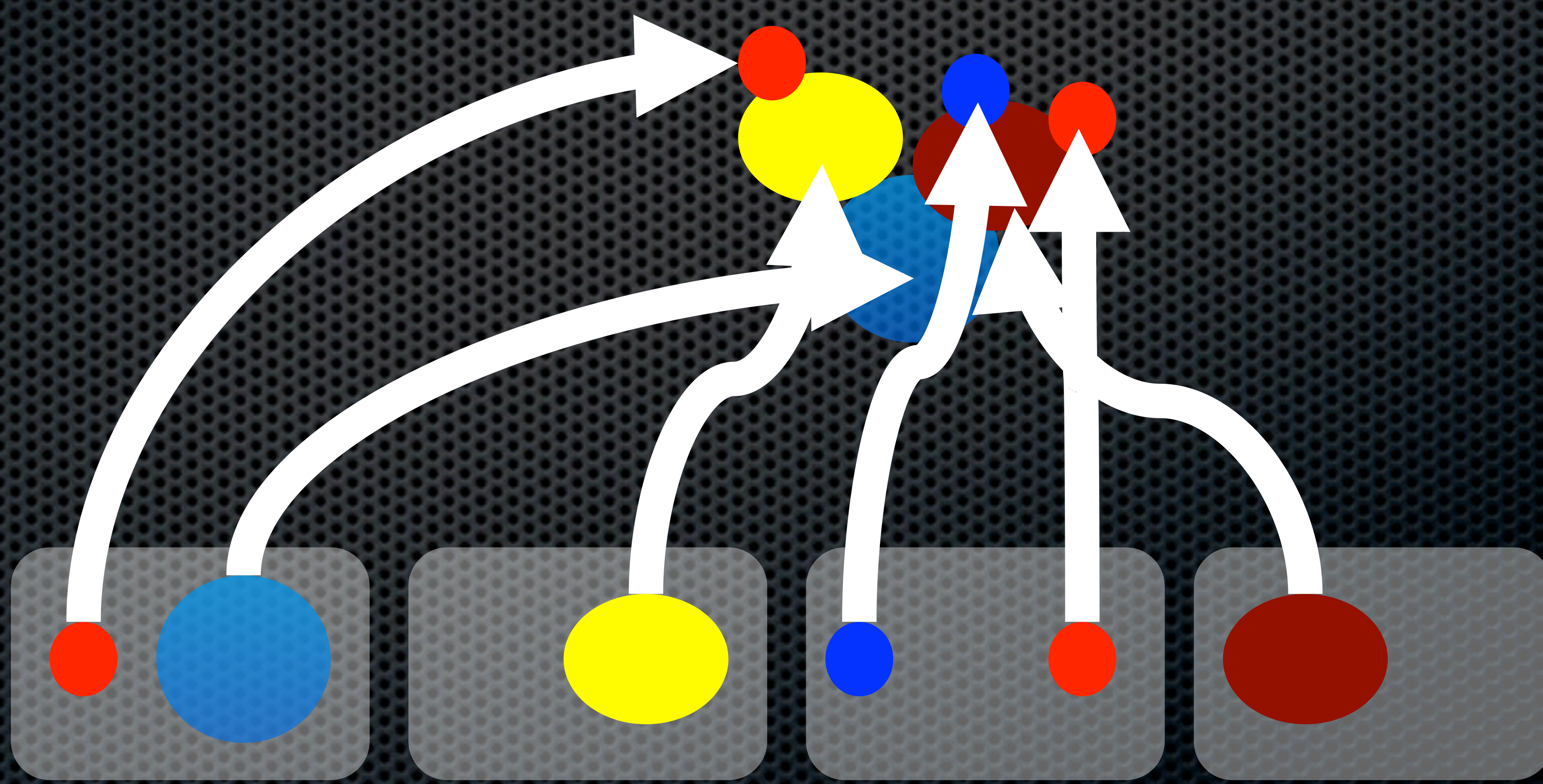
Conclusion

Data warehousing, OLTP and Distributed caching fields are all converging on in-memory architectures to get away from disk induced latencies.



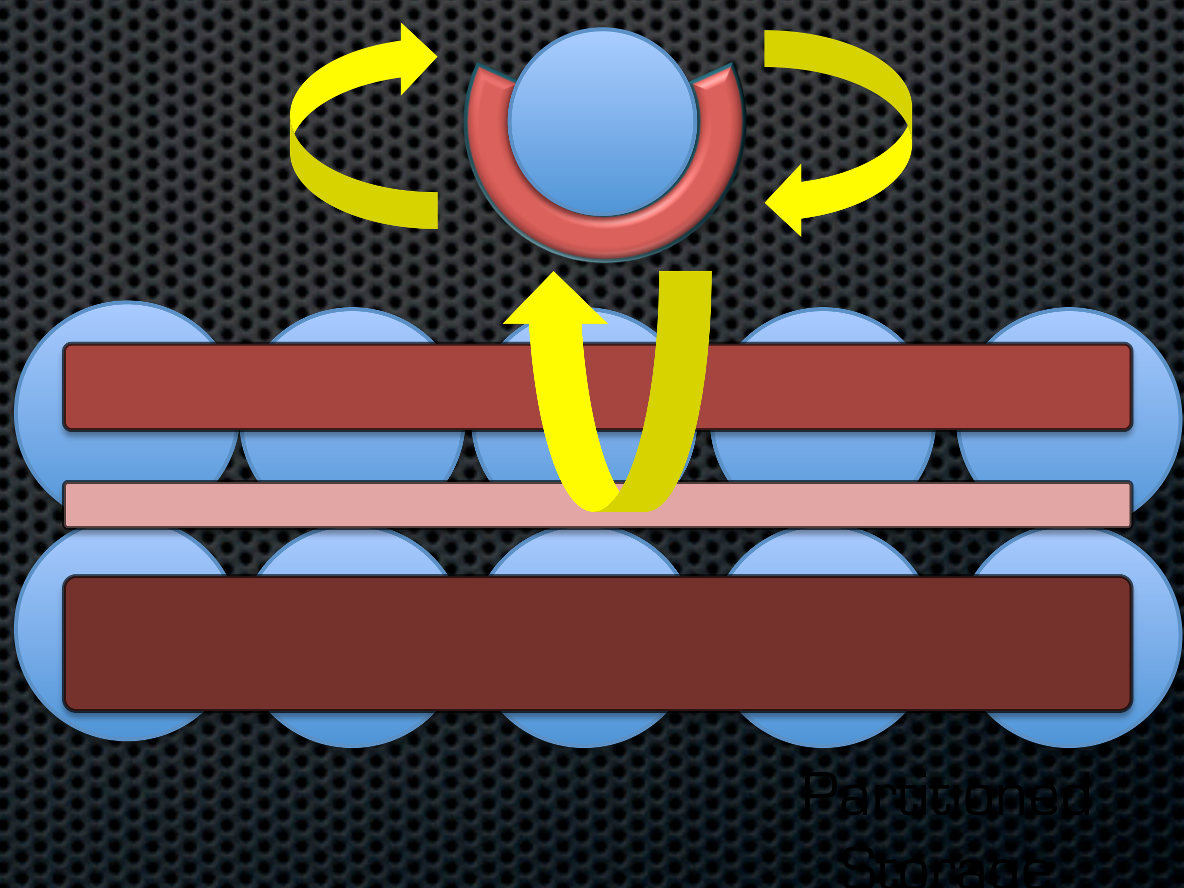
Conclusion

Shared nothing architectures are always subject to the distributed join problem if they are to retain a degree of normalisation.



Conclusion

We present a novel mechanism for avoiding the distributed join problem by using a Star Schema to define whether data should be replicated or partitioned.



Conclusion

We make the pattern applicable to ‘real’ data models by only replicating objects that are actually used: the Connected Replication pattern.

The End

- Further details online http://www.benstopford.com (linked from my Qcon bio)
- A big thanks to the team in both India and the UK who built this thing.
- Questions?