Linkedinfrastructure

QCON 2007

Jean-Luc Vaillant, CTO & Co-founder

Disclaimer: I will NOT talk about...

- Communication System
- Network Update Feed
- Authentication
- Security
- Payment
- Tracking
- Advertising
- Media Store
- Monitoring
- Load Balancing
- etc

Quick Facts about LinkedIn

- Founded end of 2002, based in Mountain View
- Website opened to the public in May 2003
- Membership:
  - 16M members, > 1M new members per month
  - 50% membership outside the US
  - > 30% users active monthly
- Business:
  - Profitable since 2006
  - 180 employees (and yes, we're hiring!)

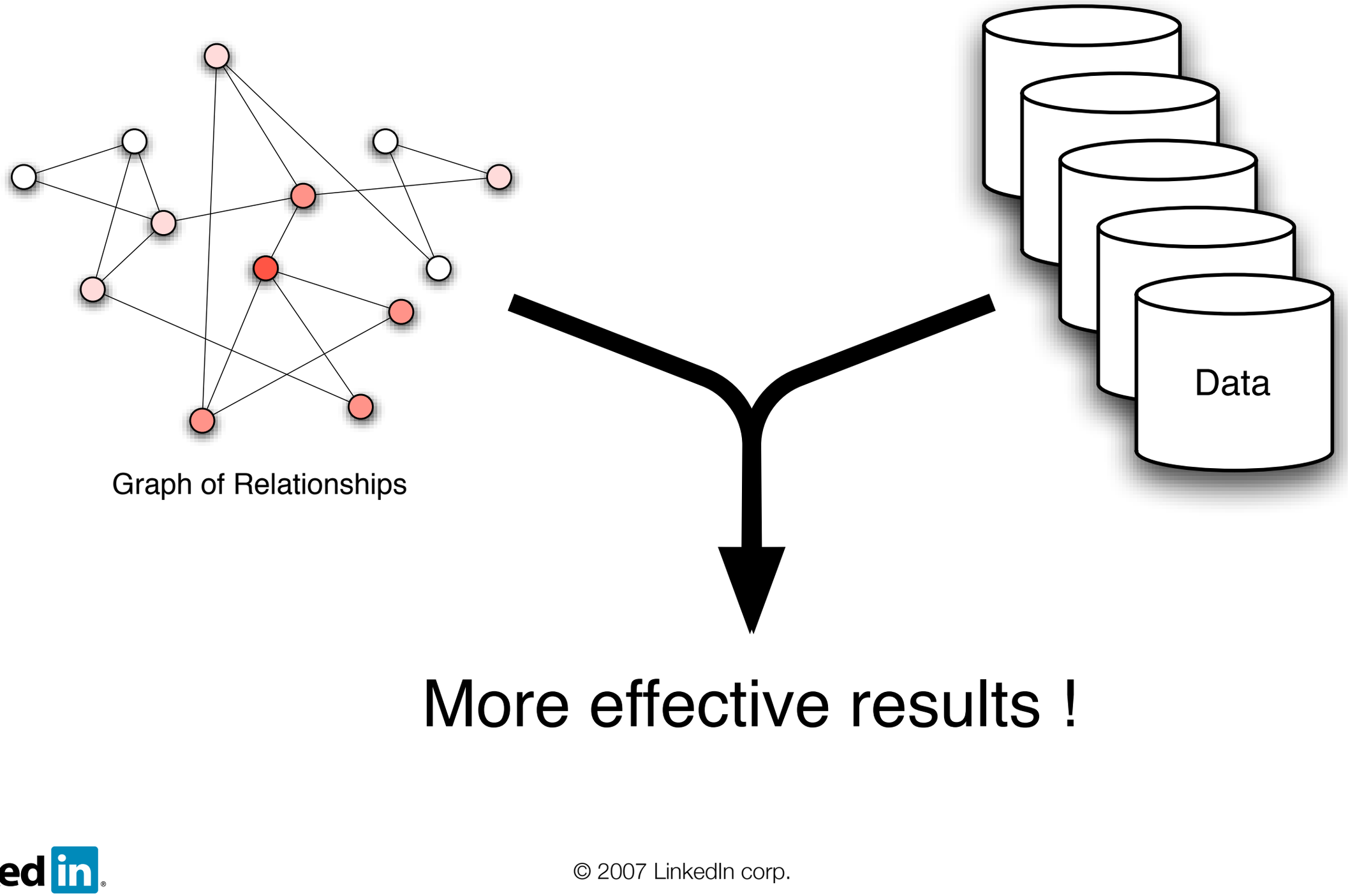
Technical stack

- Java! (a bit of Ruby and C++)
- Unix: production is Solaris (mostly x86), development on OS X
- Oracle 10g for critical DBs, some MySQL
- Spring
- ActiveMQ
- Tomcat & Jetty
- Lucene

The “big idea”...

- Professional centric social network
- Using relationships and degree distance to:
    - improve connect-ability
    - improve relevance
    - enhance sense of reputation
- “Your world is NOT flat”, relationships matter!

The “big idea” (cont)



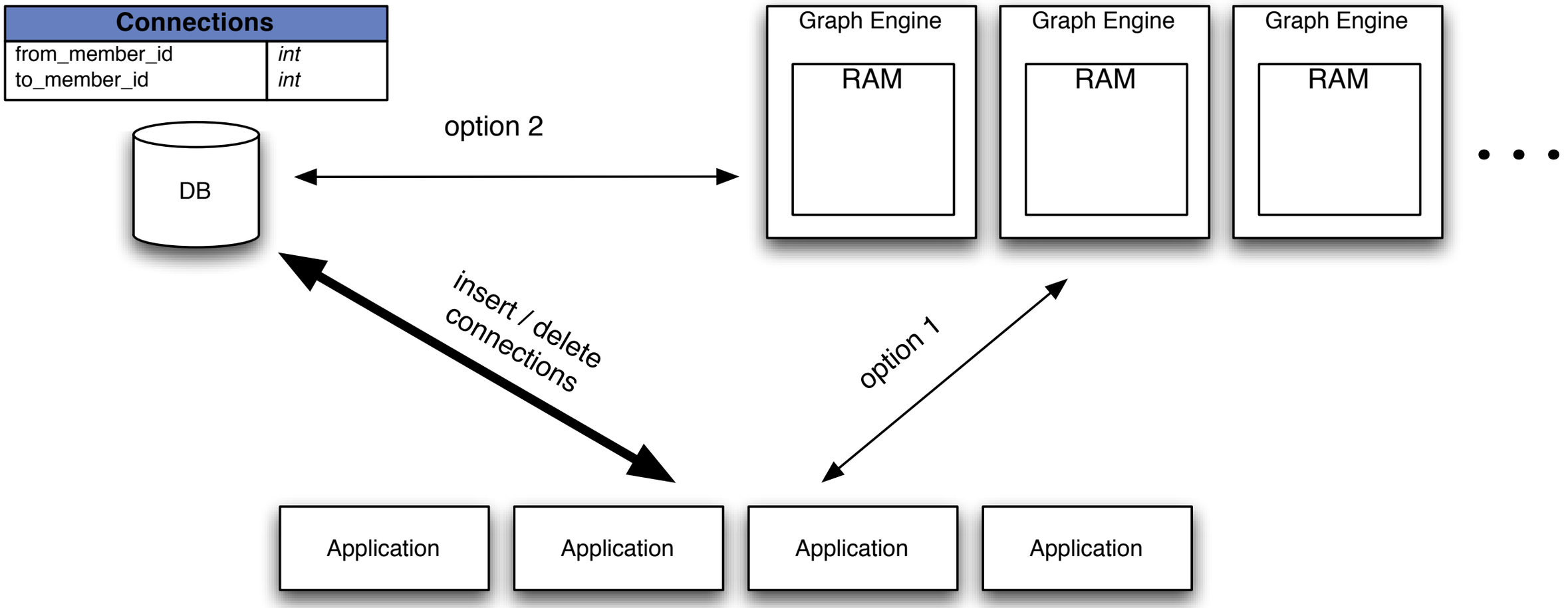
More effective results !

Just a little problem...

- Typical graph functions:
  - visit all members within N degrees from me
  - visit all members that appeared within N degrees from me since timestamp
  - compute distance and paths between 2 members
- Graph computations do not perform very well in a relational DB :-(
- Only way to get the performance needed is to run the algorithms on data stored in RAM!

Just a little problem...

Q: how do we keep the RAM DB in sync at ALL times?



Sync option 1

- Application updates DB and informs graph engines of the change...
  - direct RPC:

```
db.addConnection(m1,m2);
foreach (graphEngine in GraphEngineList) {
     graphEngine.addConnection(m1,m2);
}
```

    - reliable multicast: better but system is very sensitive to network topology, latency and availability
    - JMS topic with durable subscription
      - messaging system becomes second POF
      - provisioning new clients becomes much more painful
- if application “forgets” to send the second notification we’re in BIG trouble
- Fairly “typical” 2 phase commit problem :-(

Sync option 2

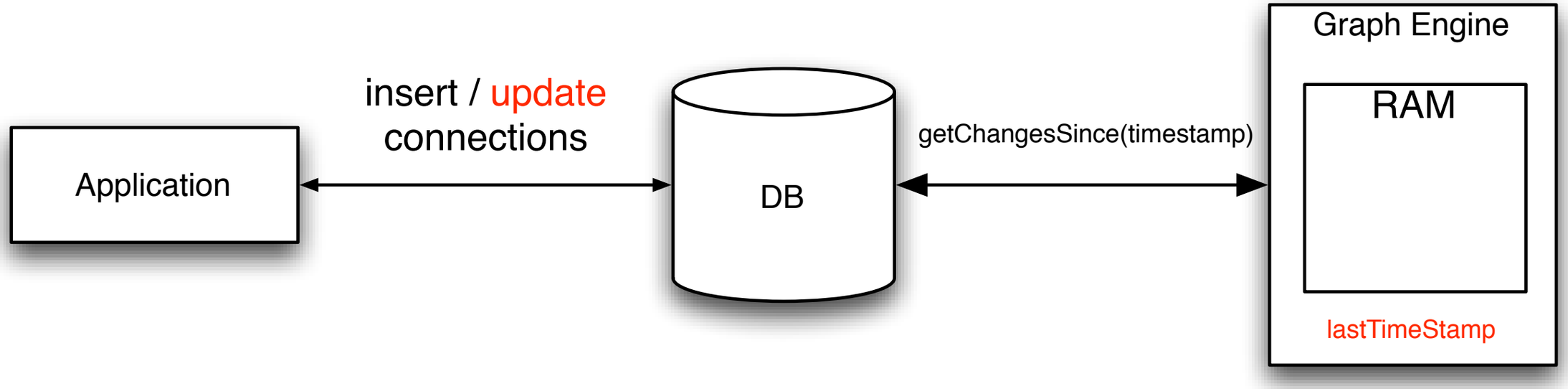
- Application updates DB (as usual) and graph engines manage to replicate the changes made in the DB
- Provides isolation of concerns
- Technology exists, it’s called DB replication
- There is a catch, it’s proprietary, opaque (Oracle) and works DB to DB only
- We need a “user space” replication...

Ideal “Databus”

- Does not miss ANY event
- Does not add any new point of failure
- Scales well
- Works over high latency networks
- Does not assume that clients have 100% uptime.
  Client must be able to catch up when it is back online.
- Easy to add new clients
- Easy to use on the application side
- Easy to set up on the DB side

First attempt... using timestamps

Connections	
from_member_id	*int*
to_member_id	*int*
is_active	*boolean*
last_modified	*timestamp*



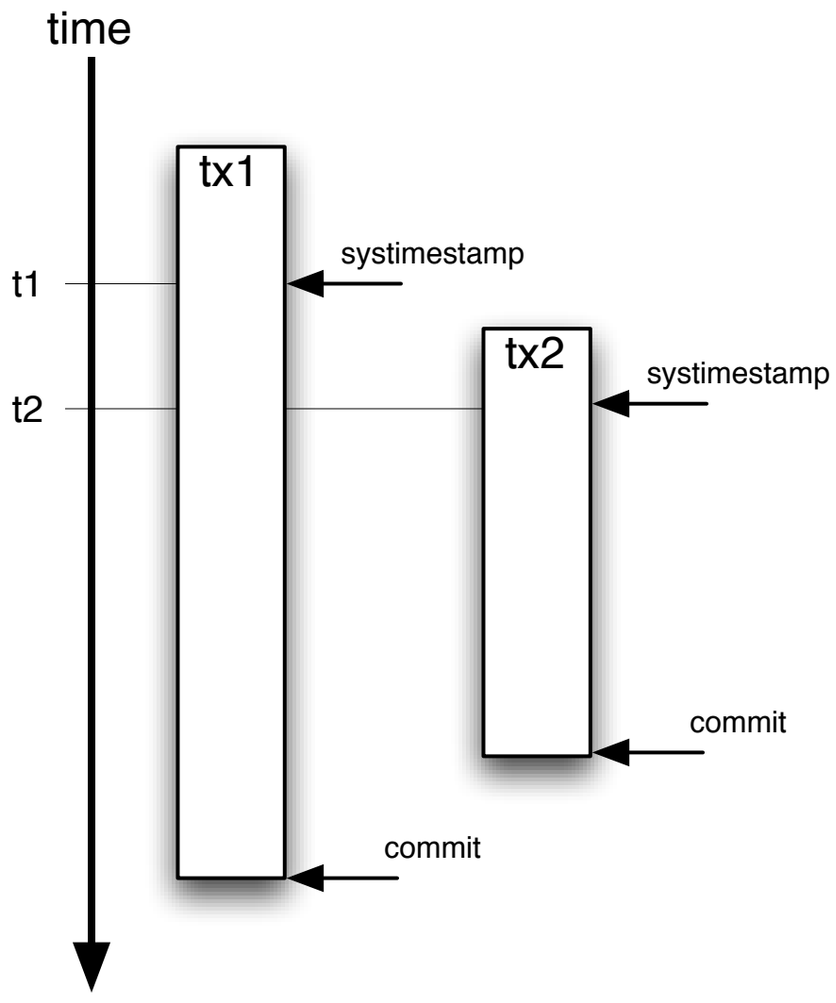
trigger on insert/update sets timestamp = `SysTimeStamp`

Works pretty well...

- DB provides full ACID behavior, no additional transaction needed
- Adding trigger and column is straightforward
- Performance on DB not seriously affected
- Client clock is not a factor
- Fairly easy for client to connect and get the event stream

A big problem though...

- Timestamps (in Oracle) are set at the time of the call, NOT at the time of commit



- tx2 commits BEFORE tx1 but t2 > t1
- if sync occurs between the 2 commits lastTimeStamp is t2 and **tx1 is missed**!
- Workaround (ugly): reread events since lastTimeStamp - *n* seconds
- Any locking and we can still miss an event

Second attempt... ora_rowscn

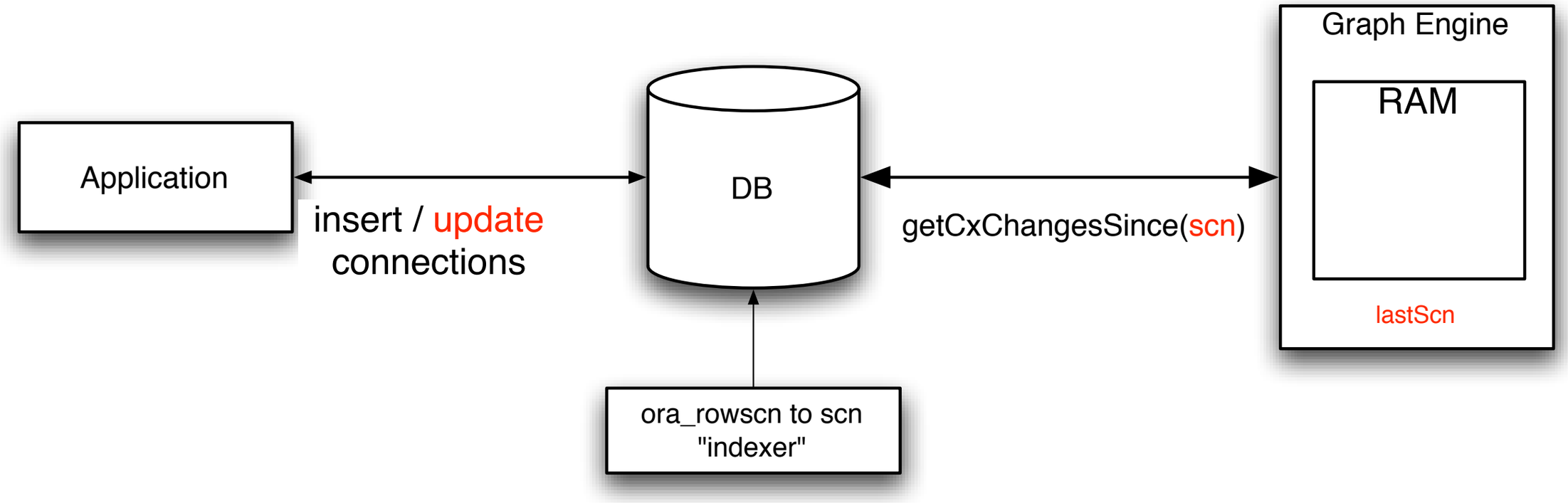
- little known feature introduced in Oracle 10g
- the ora_rowscn pseudo column contains the internal Oracle clock (SCN - System Change Number) at COMMIT time!
- by default, the ora_rowscn is block-level granularity
- to get row-level granularity: create table T *rowdependencies ...*
- SCN always advances, never goes back!
- no performance impact (that we could tell)
- little catch: cannot be indexed !

Indexing ora_rowscn?

- add an indexed column: scn
- set default value for scn to "*infinity*"
- after commit ora_rowscn is set
- select * from T where scn > :1 **AND** ora_rowscn > :1
- every so often, update T set scn = ora_rowscn where scn = *infinity*

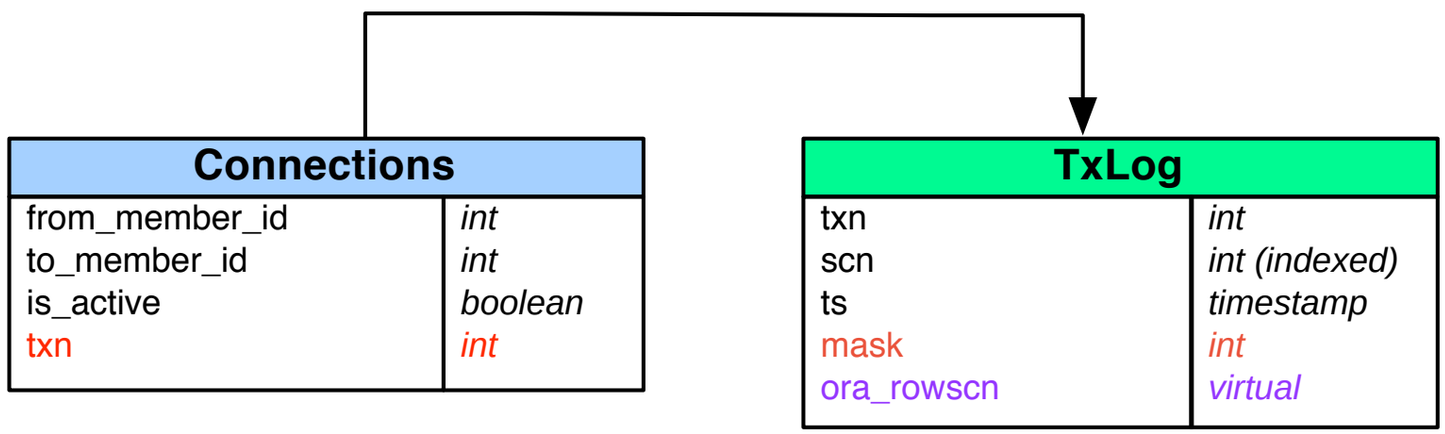
Using ora_rowscn (cont)

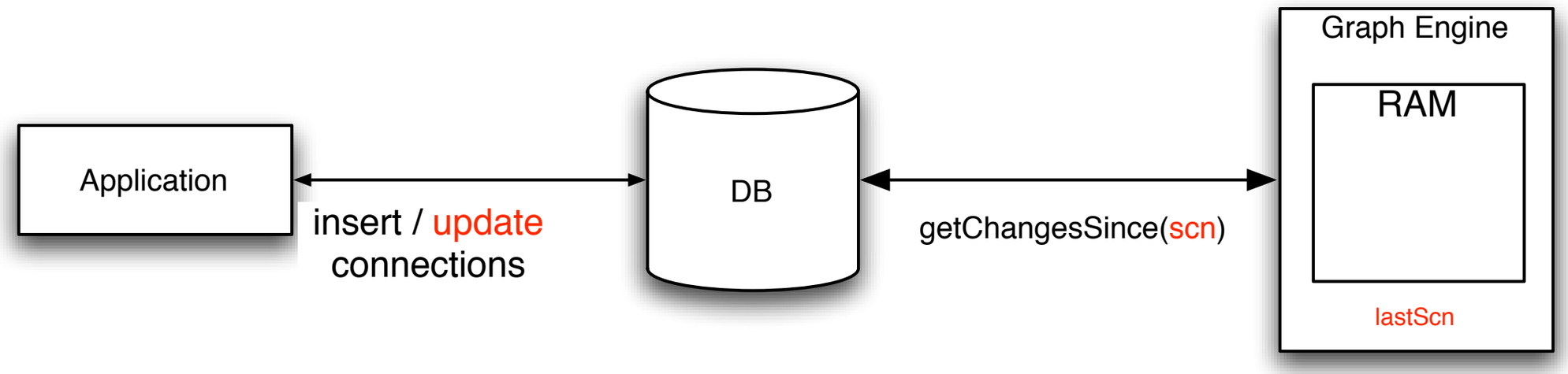
Connections	
from_member_id	*int*
to_member_id	*int*
is_active	*boolean*
scn	*int (indexed)*
ora_rowscn	*virtual*



Supporting more tables/events

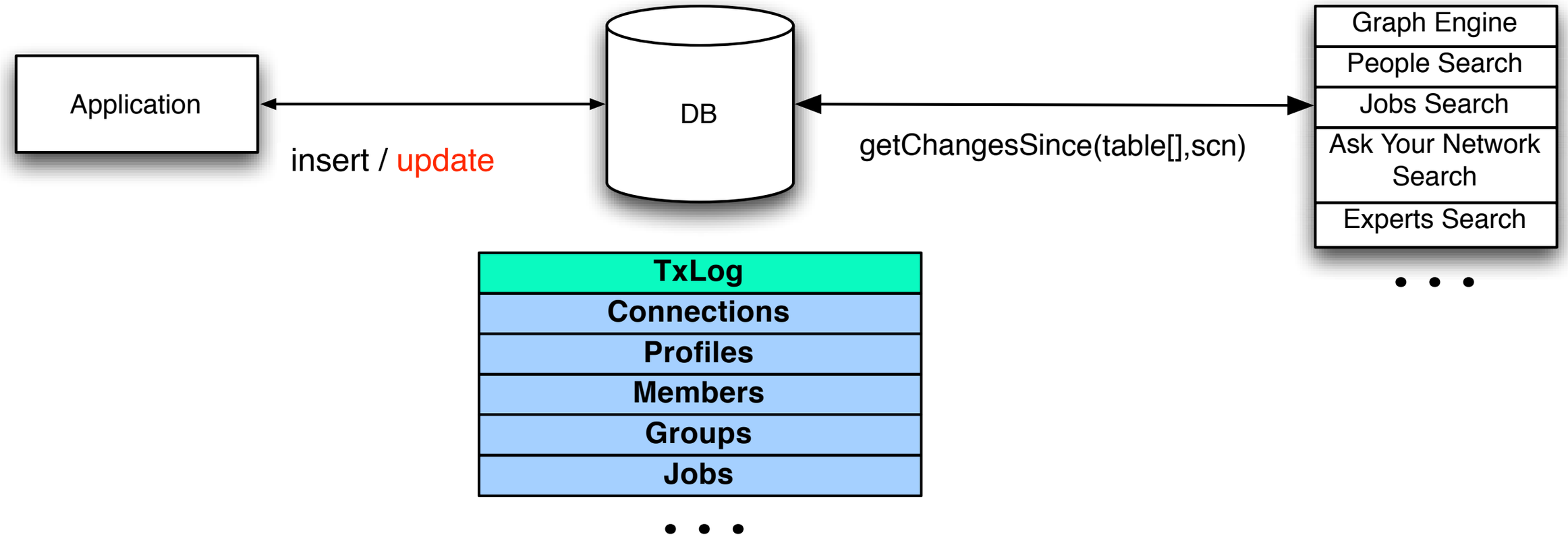
trigger on insert/update allocates txn from sequence
and inserts into TxLog





Now we have a single log for all the tables that we want to track

Many tables, many clients..

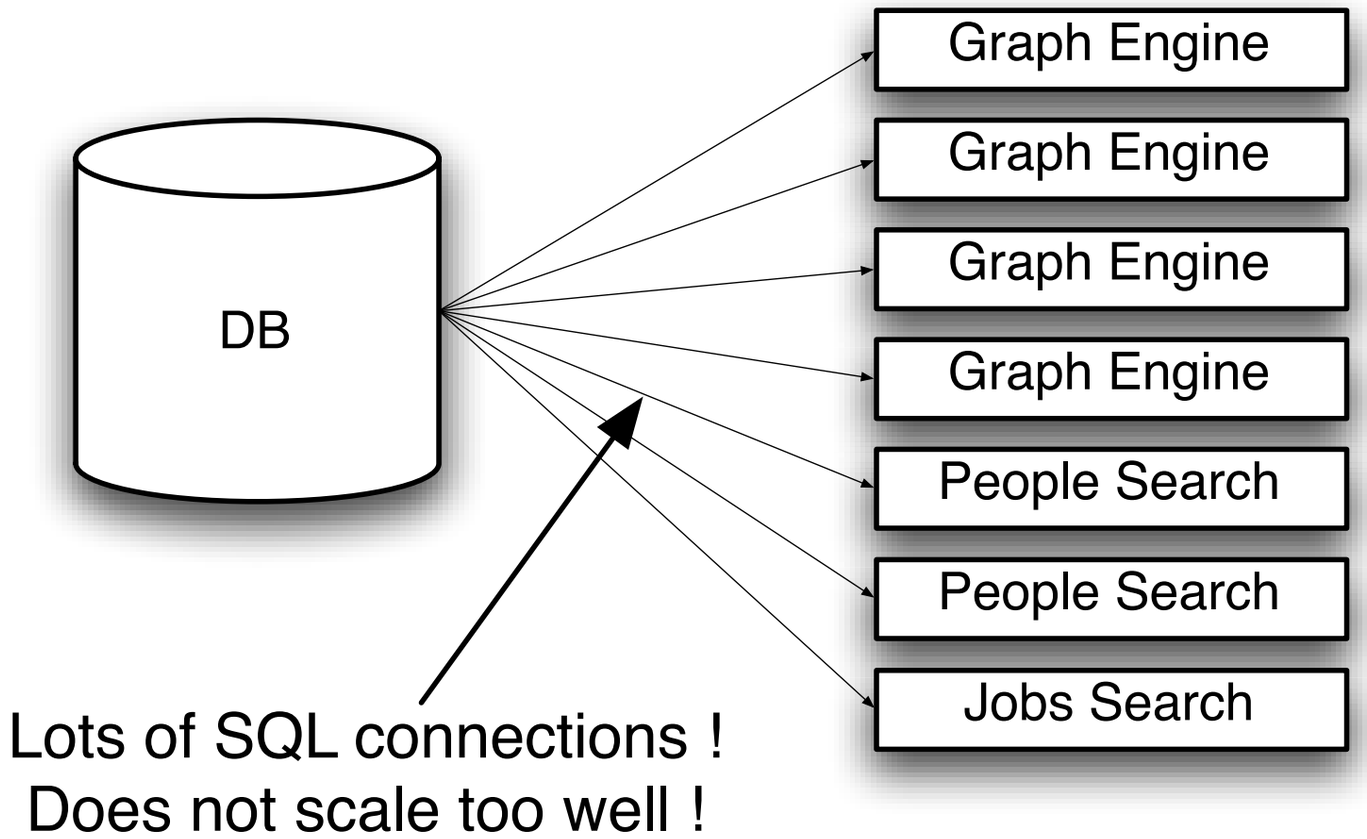


- adding a new “databusified” table is (fairly) easy
- clients can track many tables at once
- transactions involving many tables are supported

Ideal “Databus” ?

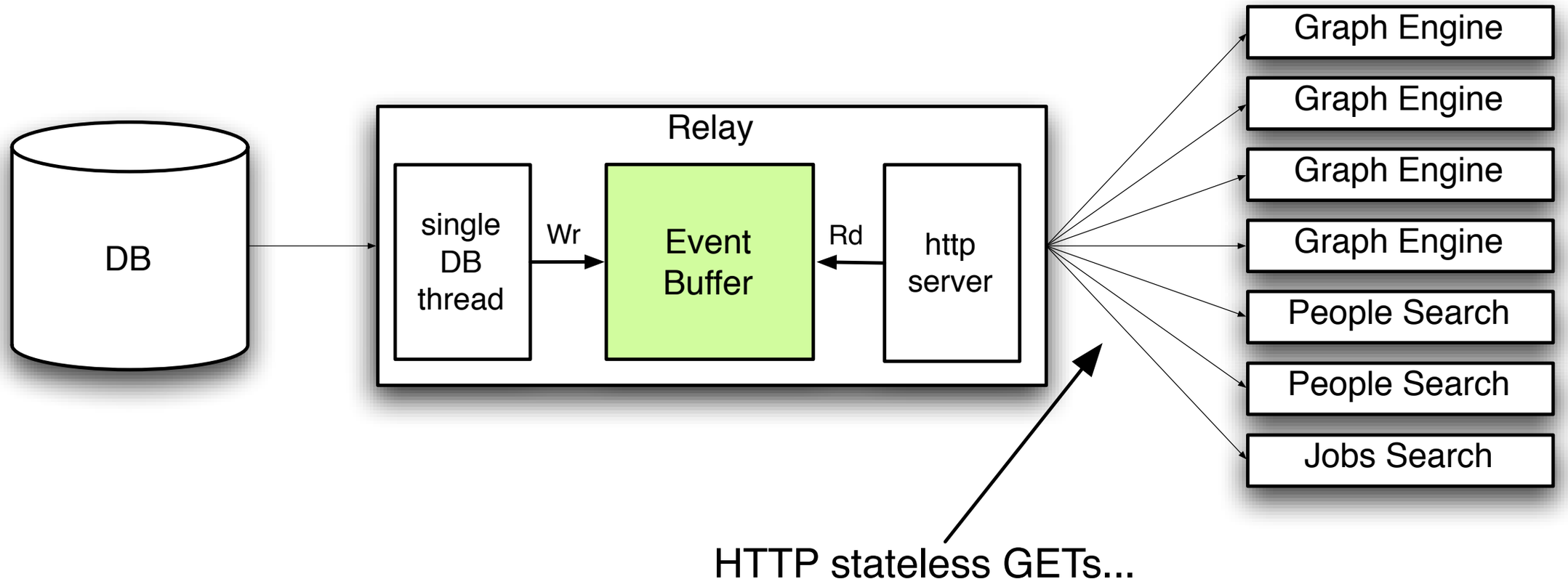
- Does not miss ANY event
- Does not add any new point of failure
- **Scales well?**
- Works over high latency networks
- Does not assume all clients to be 100%
- Easy to add new clients
- Easy to use on the application side
- Easy to set up on the DB side

Scaling?



Adding a lot of clients will eventually kill my databases

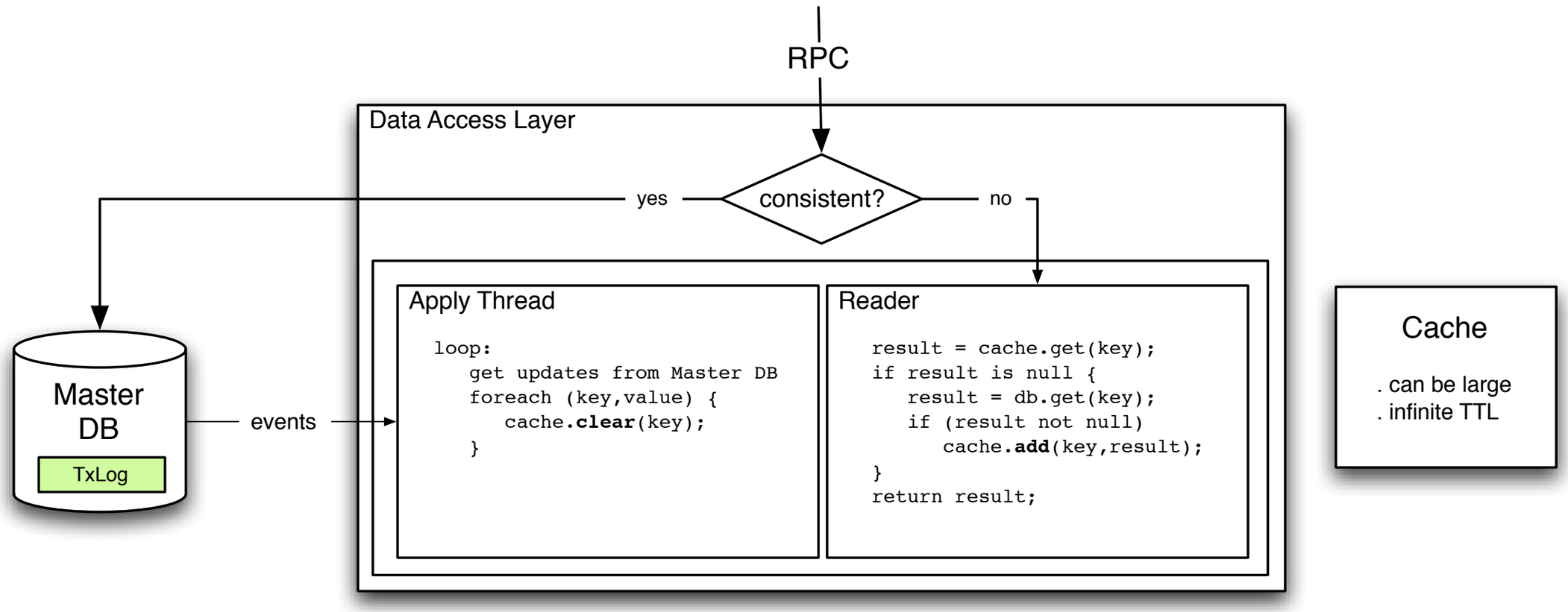
Scaling!



- only a few DB clients (for HA)
- hundreds of clients load balanced over HTTP
- relays can be cascaded (within reason)
- clients do not talk to DB at all!

Additional applications...

- Cache invalidation

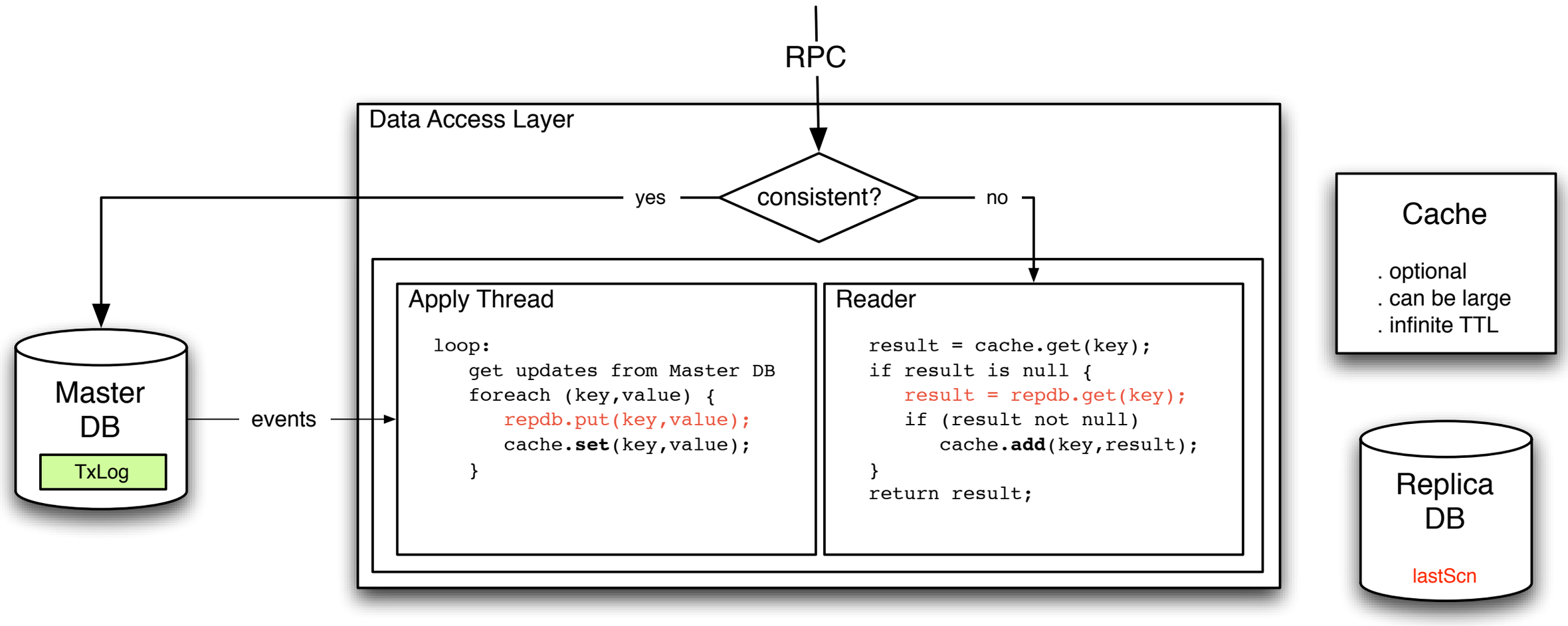


consistent = write operations or any subsequent read within N seconds...

note: N can be dynamically computed and adjusted for latency

Combination caching & replication

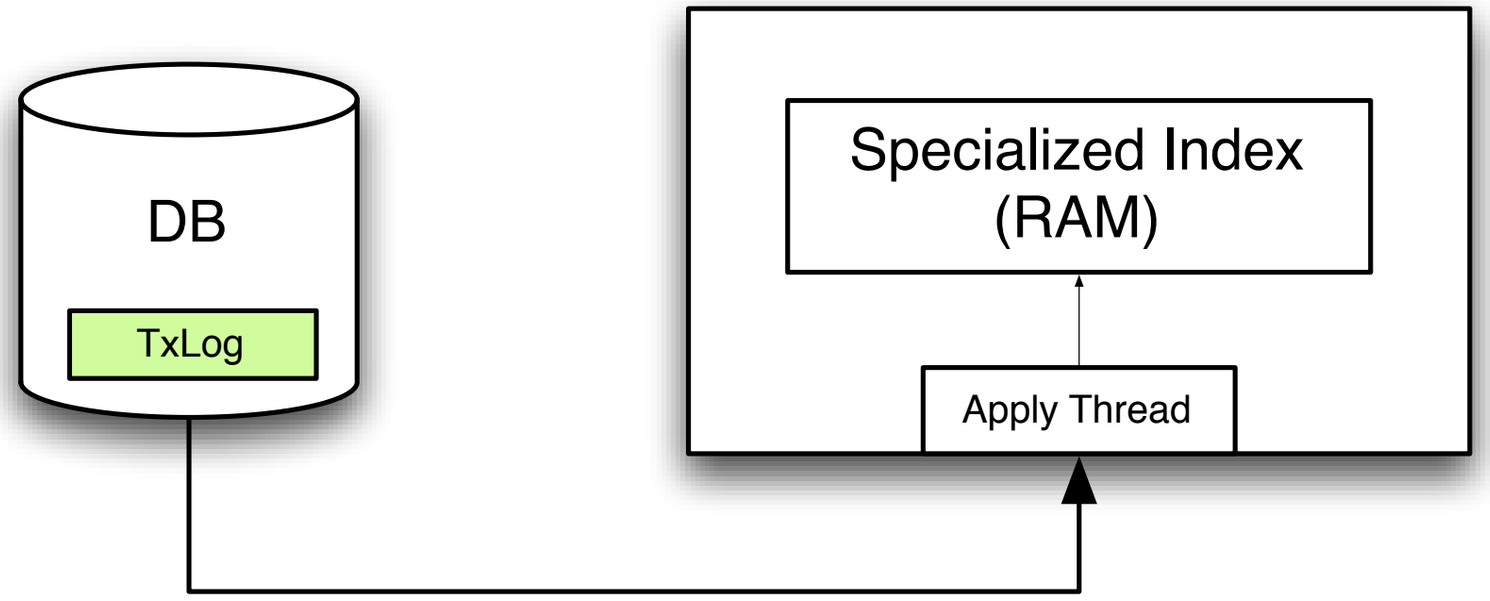
- Even better, if cache is empty, fill it up from a LOCAL replica



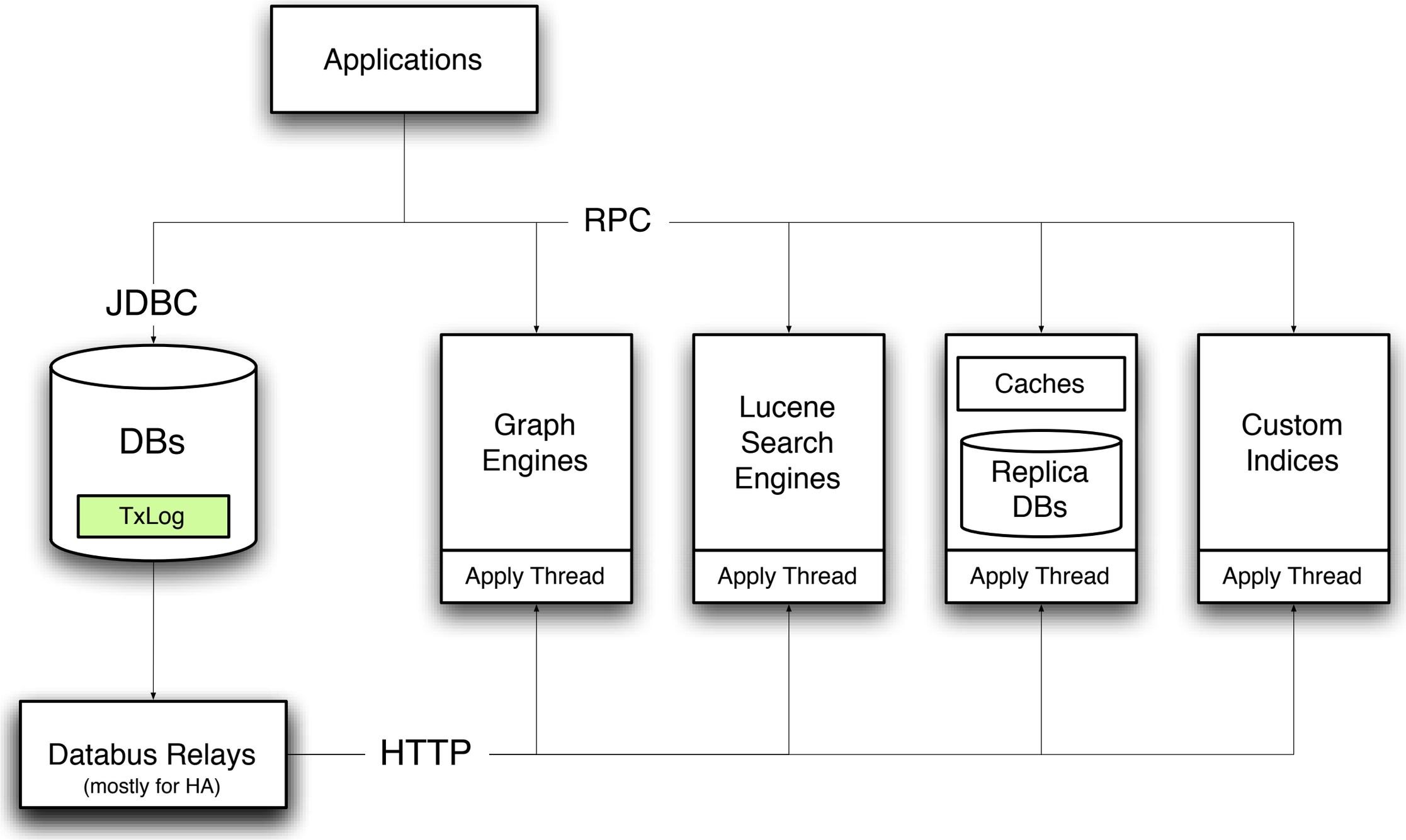
Some race conditions exists, details for dealing with them are omitted

Specialized indices

- bulk lookups: “which of these 800 contacts in my address book is a member?”
- spatial indices...
- partial indices for time relevant data (most recent, short lived data)



The overall view...



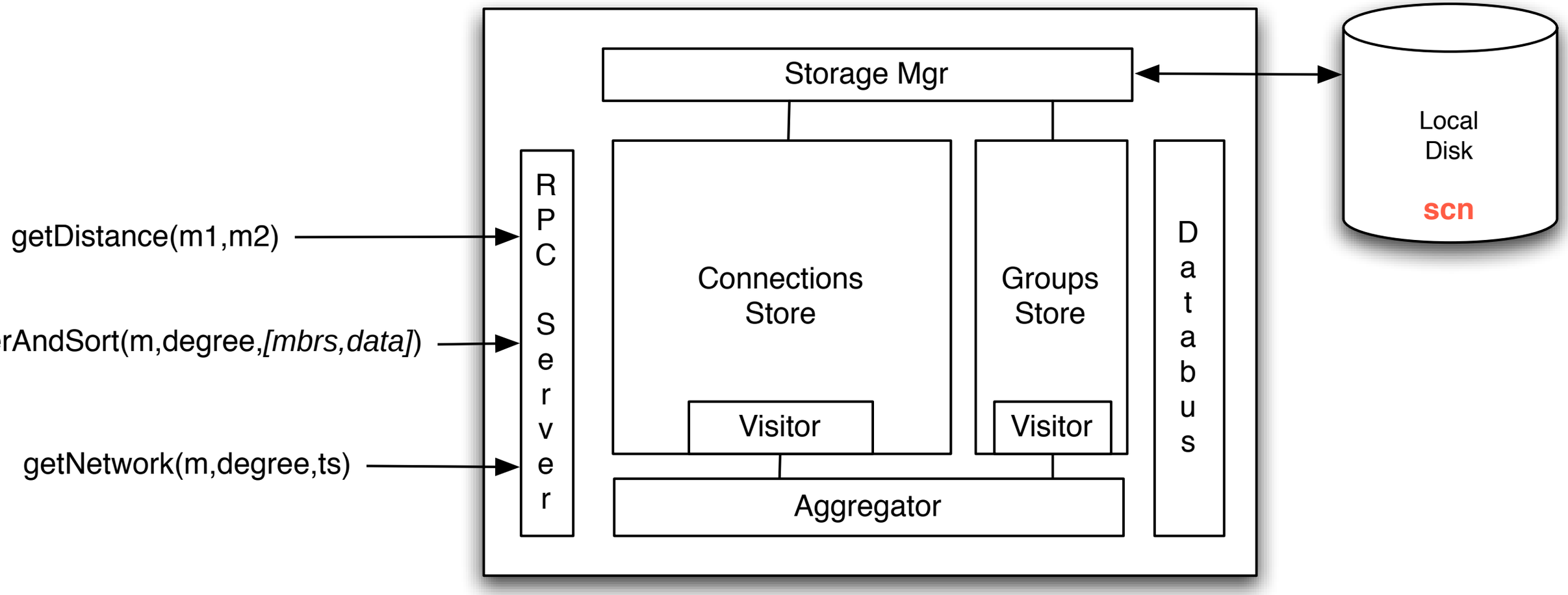
Vendor lock-in?

- Our capture code is indeed Oracle specific but...
- All we need from the DB is
  - having a unique ID per transaction (txn). No ordering required, just unicity
  - having ONE table with a non indexable “SCN”
  - serializable transactions
  - (optional) alert mechanism (cond.signal())

Databus summary

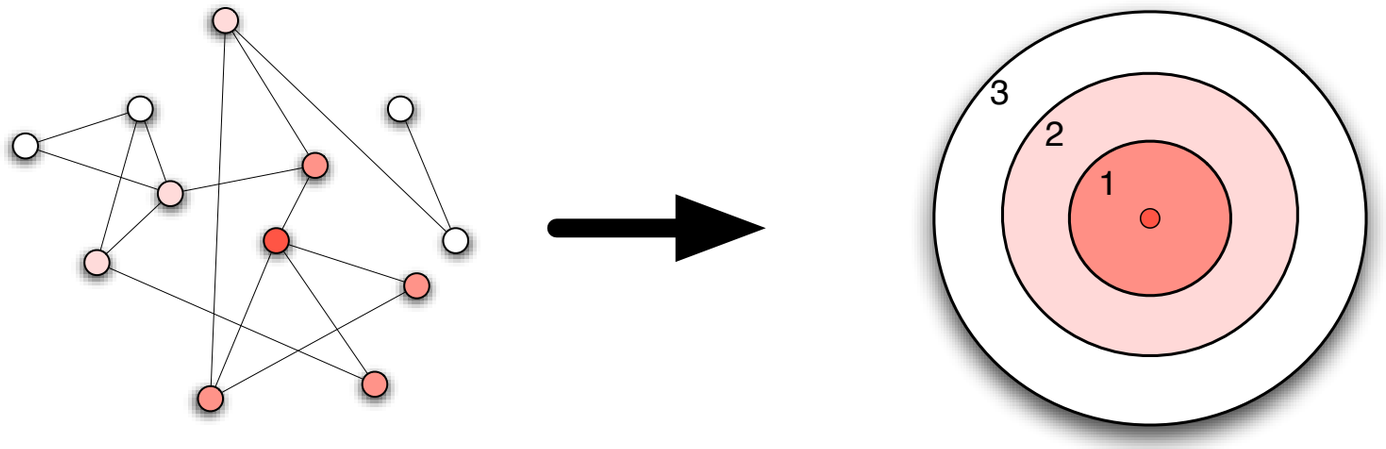
- it is NOT a messaging system, it only provides the last state of an object
- it is NOT a caching system or datastore, it provides changes "since a point in time", not "by key"
- it will provide 100% consistency between systems... eventually...
- no additional SPOF: it's you commit to the DB, it will be available on the bus
- moves replication benefits out of the DBA land into the application space
- very limited support from the DBMS required
- plugging in the Databus from a SWE POV is as simple as wiring a Spring component and writing an event handler...
- allows for a full range of data management schemes if the data is mostly read

Back to the Graph Engine...



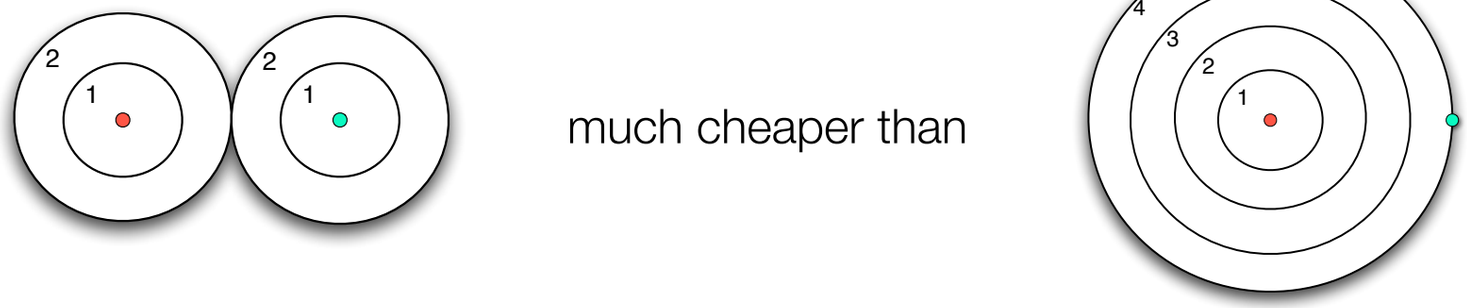
Algorithms

- visitor pattern within a “Read Transaction”, standard Breadth First



- a little fancier for visitNewSince(TimeStamp ts) if you want to avoid traversing the graph twice

- use symmetry when possible



First challenge: Read Write Lock gotchas

- Wikipedia:

  A **read/write lock pattern** is a software design pattern that allows concurrent read access to an object but requires exclusive access for write operations. In this pattern, multiple readers can read the data in parallel but needs exclusive lock while writing the data. When writer is writing the data, readers will be blocked until writer is finished writing.

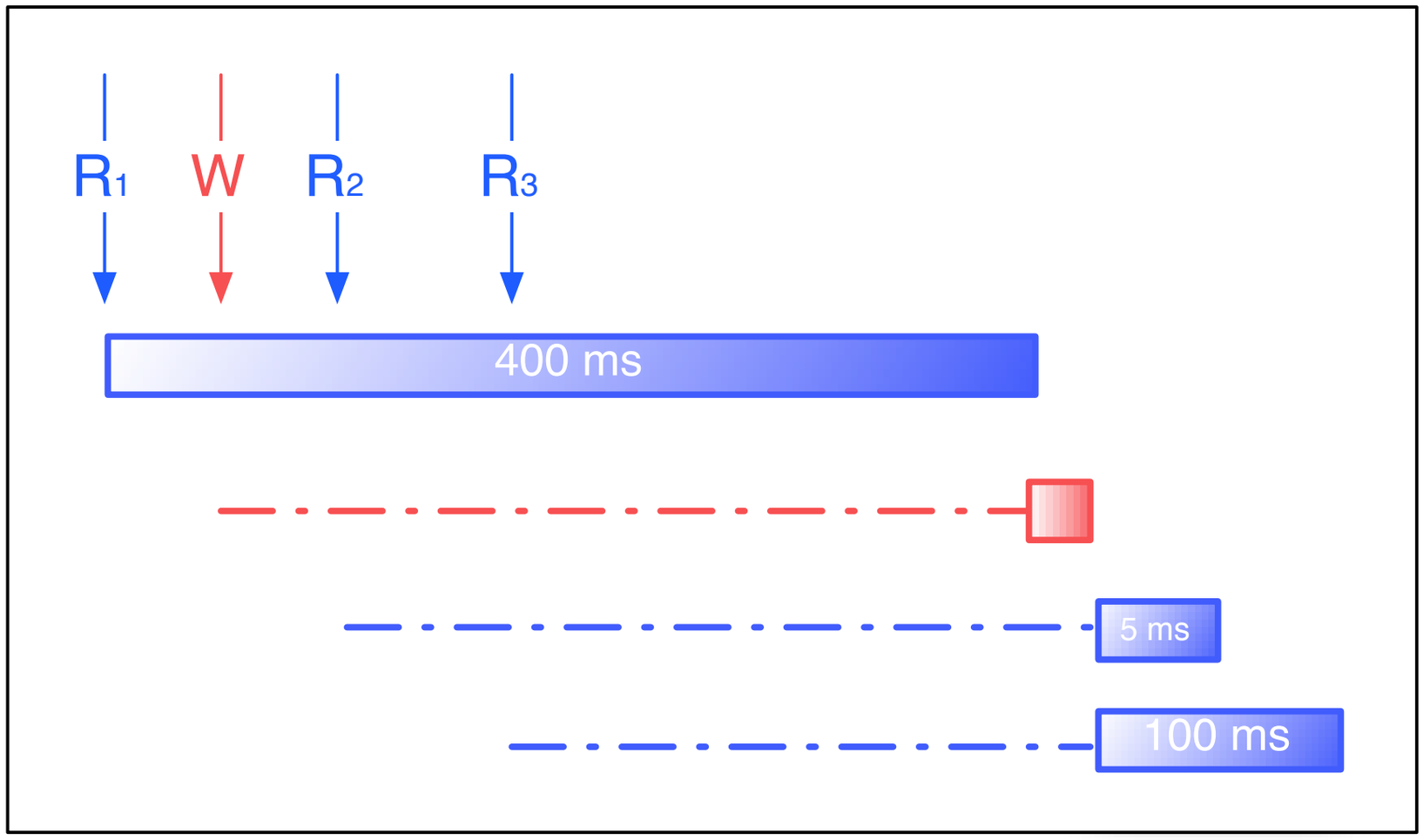
- Sounds like a great locking mechanism for any structure with a LOT of reads and a FEW tiny (serializable) writes...

- Graph index is one of these...

- So why do we observe?
  - CPU caps way under 100%
  - Graph RPC calls take much longer than expected

Read Write Lock gotchas

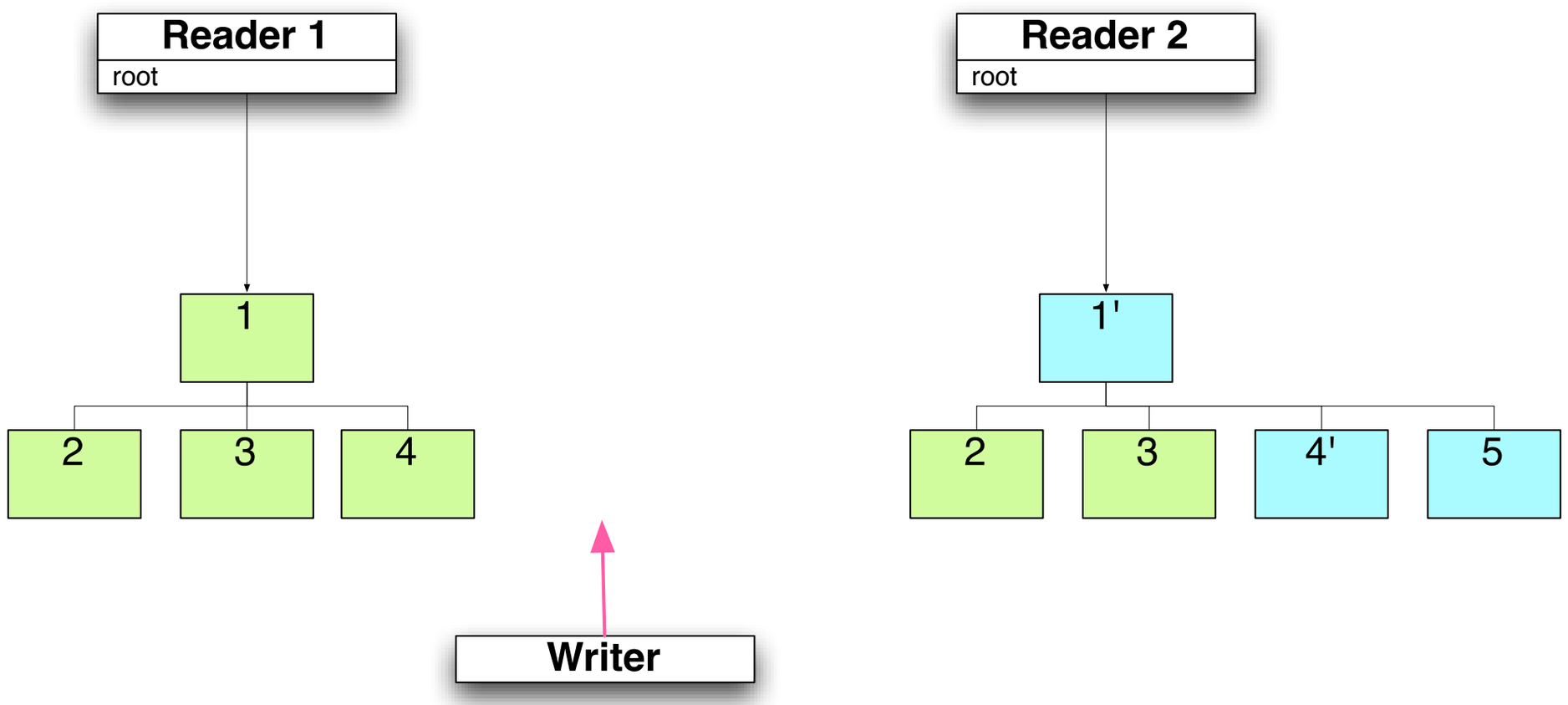
Problem occurs when some readers take significant time...



R2 is delayed by nearly 400 ms for a 5 ms call!

Read Write Lock gotchas

- Solution: don’t use RW lock, use Copy On Write...

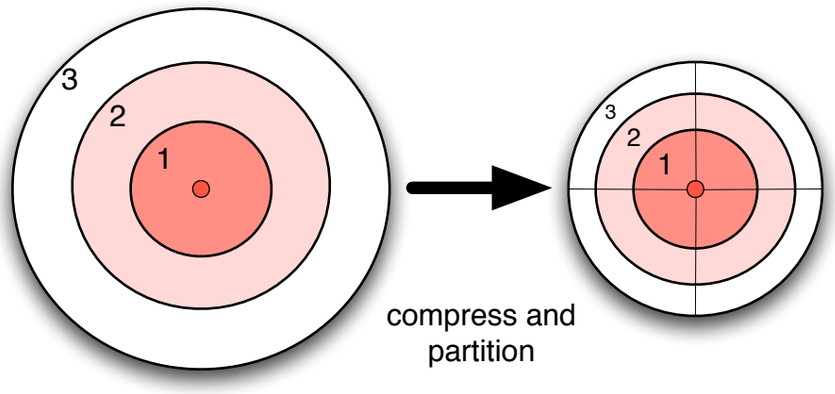



```
add node 5
dup node 4 and 1
switch root
wait for previous readers to finish
free nodes 1 and 4
```

Second challenge: one's network size...

- Graph of connections is highly skewed: yet another power law...
- Heavy connectors "attract" each other which leads to a "black hole" effect
- Once you introduce distance > 1 the network size grows very fast !

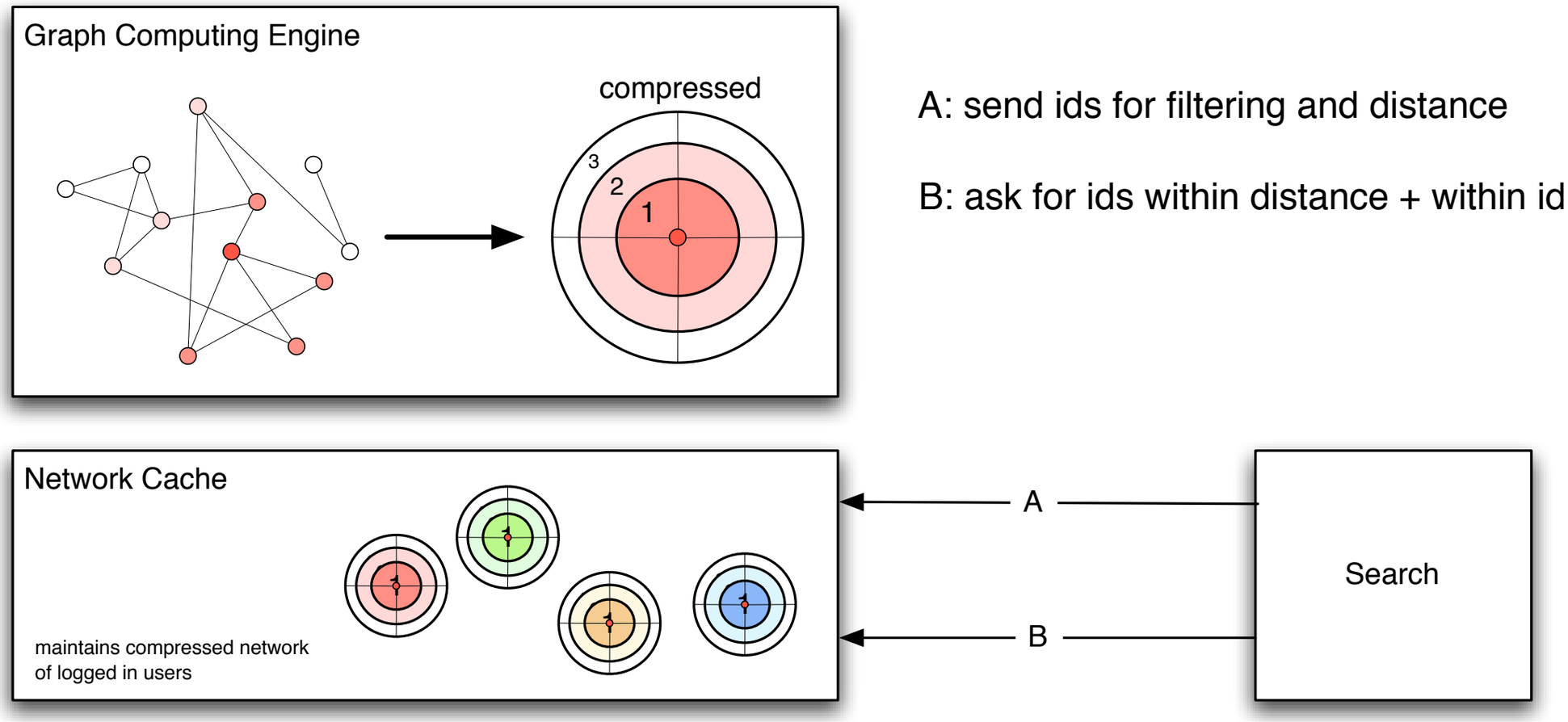
- Compressing can flatten that curve considerably...



Network artifact is now:

- cacheable
- transportable over the network
- whole, per degree, per "chunk"

With caching



A: send ids for filtering and distance

B: ask for ids within distance + within id range

- cache is maintained compressed
- compression is designed so visits, lookup and partial fetches are fast!
- network can be shipped to other systems for additional relevance

Summary

- Databus infrastructure useful for....
  - maintaining indices more efficiently than in a std relational DB
    - Graph of relationships beyond first degree
    - Text based indices (Lucene)
  - splitting data persistence from read queries
    - use DB mainly for storing data and maintaining TxLog
    - use RAM for processing/querying the data
  - keeping distributed caches in sync with Master DBs

Q & A