Lag Sucks!

Making Online Gaming Faster with NoSQL
(and without Breaking the Bank)

R.J. Lorimer
Director of Platform Engineering
Electrotank, Inc.

Robert Greene
Vice President of Technology
Versant Corporation

GAME DEVELOPERS CONFERENCE® ONLINE
AUSTIN, TX
OCTOBER 9–11, 2012
EXPO DATES: OCTOBER 9–10
2012

Electrotank, Inc.

es5™ electroserver

Scalable Socket Server

Tested to 330,000 Users
on a Single Server

MMO Game Platform

Rapid Development APIs
Avatars, Questing, Achievements, Inventory, etc.

Mattel, Ubisoft, FunGoPlay

eUP™ electrotank Universe Platform

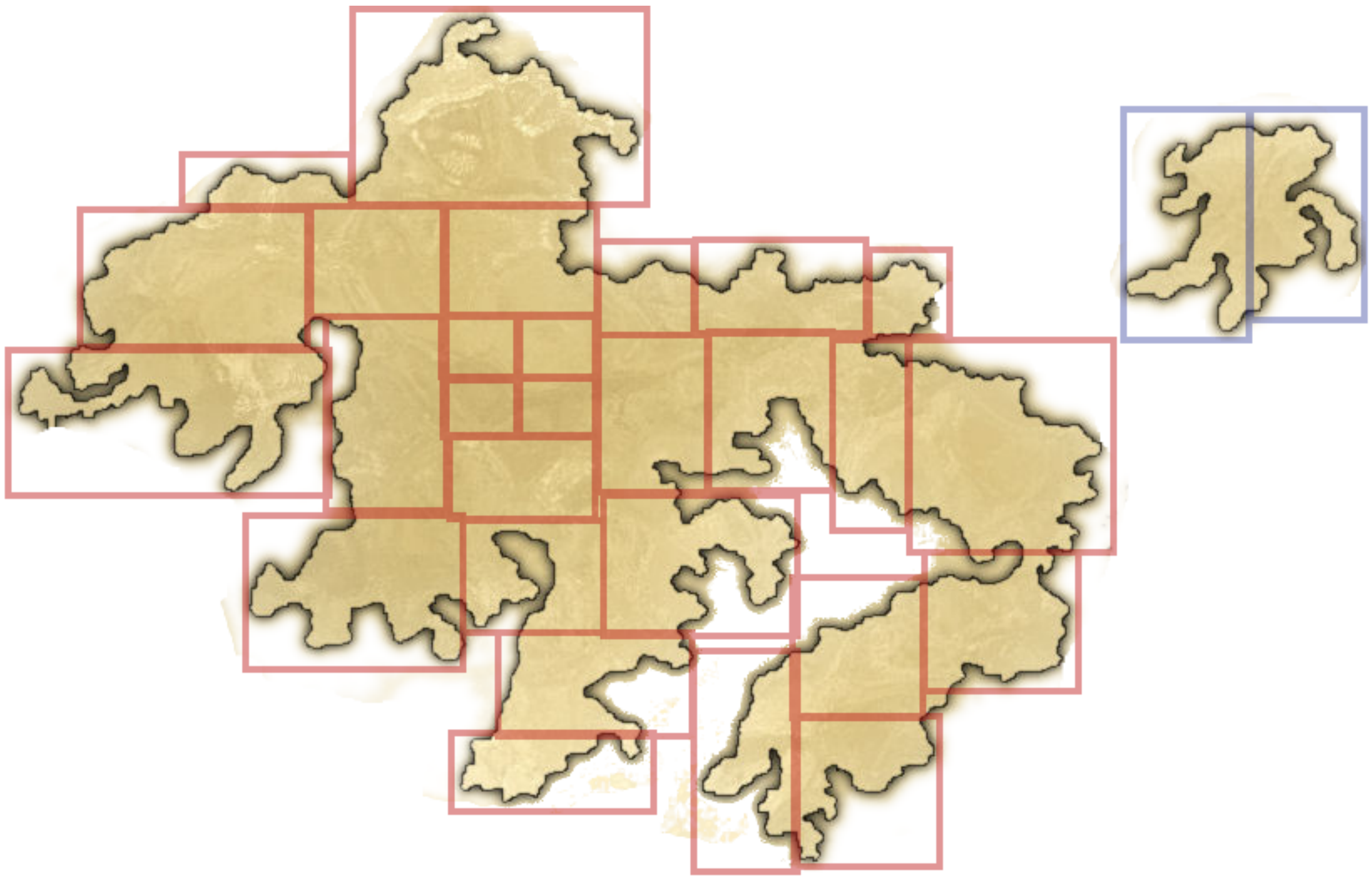
Modeling Worlds

- Diverse maps
- Clusters of activity
- Observing and interacting with:
  - Other players
  - NPCs
  - World items
- All data-driven

Modeling Worlds

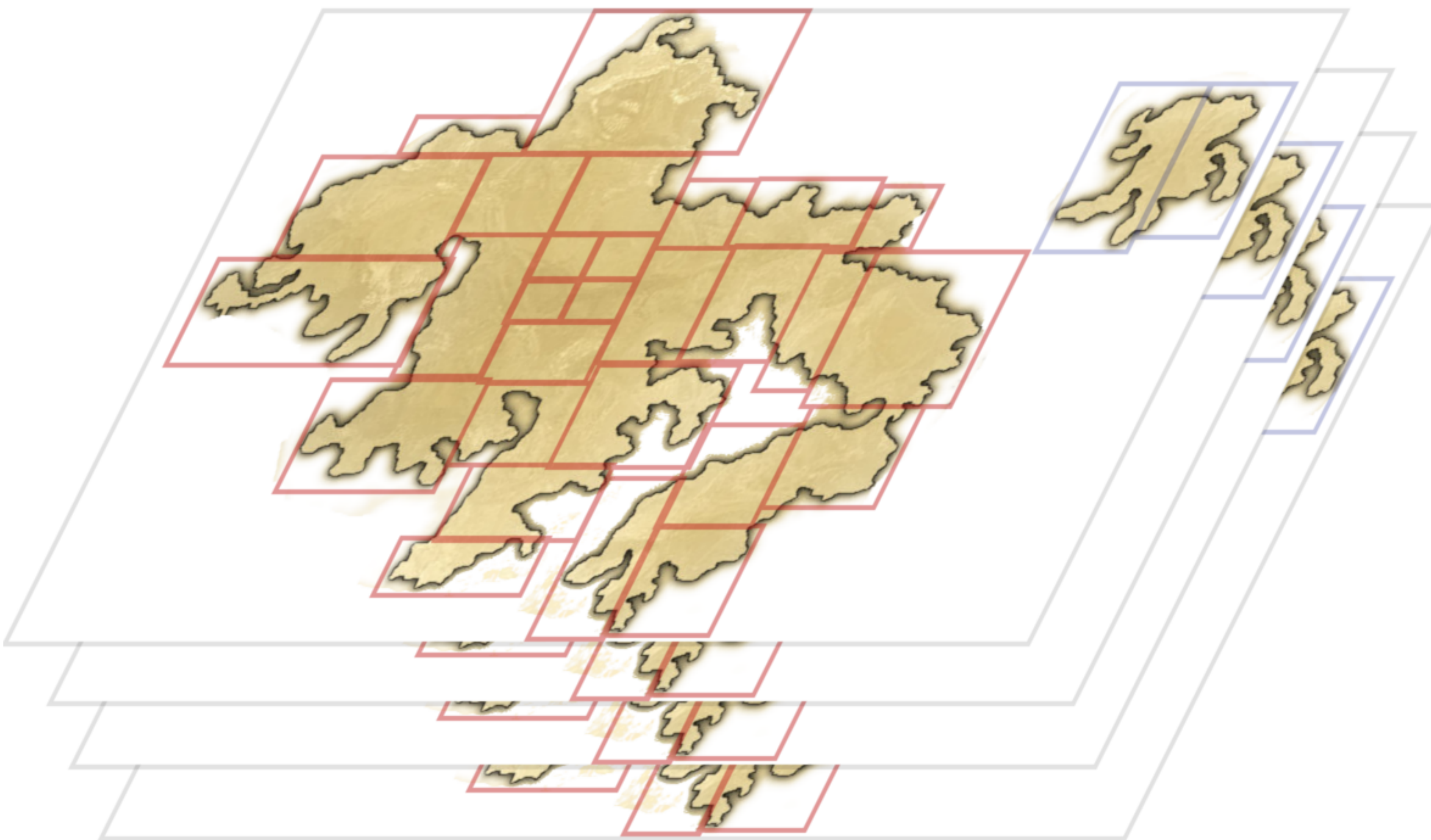
Discrete Maps are “Spaces”

Modeling Worlds



Spaces are sliced into “Regions”

Modeling Worlds

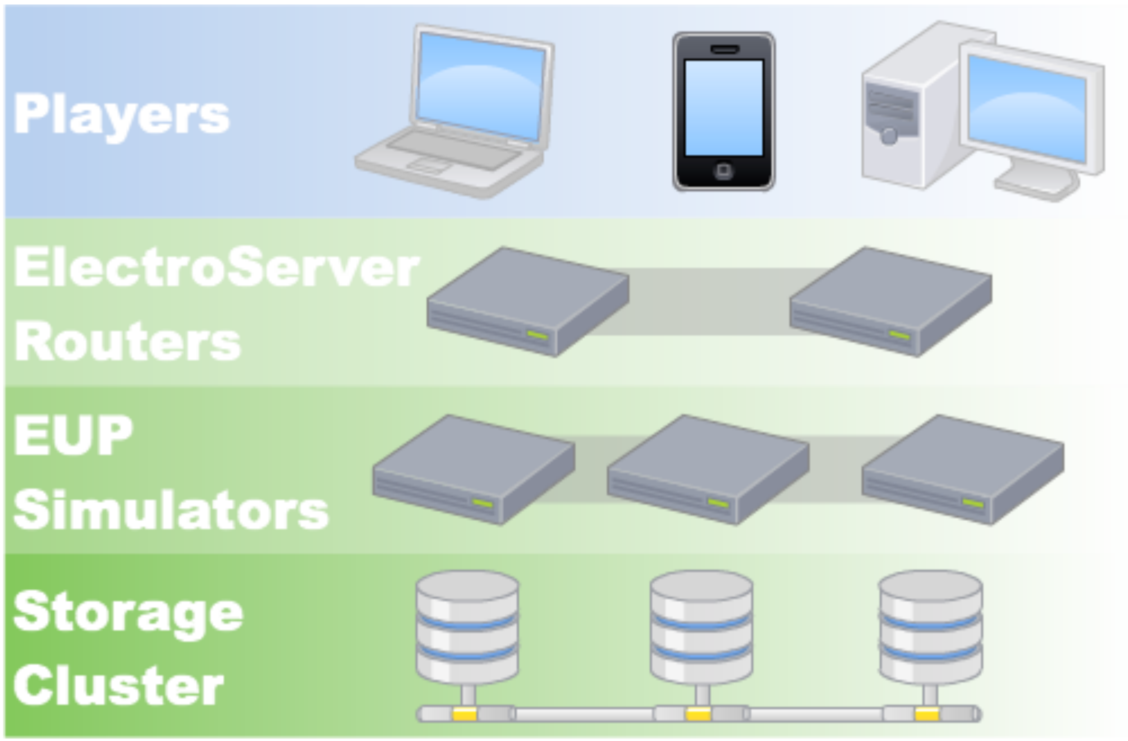


Space can be cloned into “Layers”

Modeling Worlds

- Spaces may be very big
- Player’s view is very small: “Awareness Bubble”
- Players can see local region objects, and also local players
- Region data is local to the region; player data is local to the player
- Locality is key

Deployment Tiers



- Players connect to ElectroServer
- ElectroServer provides stateful connections and protocol
- Player data channeled to correct EUP simulator node
- EUP nodes run game logic and persist data
- Game logic: spatial awareness, movement, collision detection, quest evaluation, item/inventory management, etc.
- Storage cluster provides persistence. Data roughly divided as game content and player data
- Ideally each tier independently scalable

How Do We Scale It?

- Regions hosted on servers
- Player’s primary server defined by region
- Related players grouped on same box (mostly)
- Avoids having to cluster heavy simulations
- Much more efficient
- Much easier to program
- Distributes server load automatically
- Regions share data via point-to-point channels
- Players transition through channels

The Storage Cluster

- Latency is key
- Lots of relationship fetching
- Lots of little records
- Heavy contention on some data
- Very little “ad-hoc”/complex querying

Problems with RDB

- SQL is slow
- No optimized cases
- Impedance mismatch - Hide with ORM
- ORMs are complex; brittle
- Heavy dependency on cache
- If cache isn't warm, you're gonna have a bad time
- DB cache is dumb

More Servers, More Problems

- Now the cache crutch won't support our weight
- Either:
  - Tons of sync data sent from server to server or...
  - Cache is evicted constantly making it ineffective
- Keep hitting that latency wall

So What Then? NoSQL

- If RDB doesn’t work, try something else!
- Lots of options
- Tailored to solving specific problems
- Many different styles to consider
- Often not atomic: Easy to scale; harder to code.
- Variable standards/spec support
- Young technologies means rough edges
- What they are good at may not be what we need

NoSQL Options

➡ Key-Value

- Good: Optimized Cases, Simple, Easy to Scale
- Bad: Atomicity, Simple, but still impedance mismatch. Querying can be hard.

➡ Document-Oriented

- Good: Simple Communication (HTTP +REST), Scaling can be easy.
- Bad: No schema means embedded schema. Protocol is expensive. Impedance still high (we're not storing documents). Querying can be very hard.

NoSQL Options

➡ Graph Databases

- Good: Conceptually similar to objects, Atomic.
- Bad: Impedance is still high: nodes and edges vs. objects and relationships. Mentally complex, and can be hard to analyze outside of code.

Neo4j
The World's Leading Graph Database

Objectivity

Not ideally suited to this problem. Might be a great backend for something like an NPC steering engine.

Object Databases

- Inherently understand OO (inheritance, polymorphism, relationships)
- No impedance mismatch. Objects are the schema. Code and data match
- Atomic. Transparent scaling may be expensive. Managed scaling may not be
- More mature
- Good spec support (JPA)

VERSANT

db4objects BY VERSANT

ObjectDB

No More Impedance

- EUP is heavy on the "O" in ORM already
- Most common use-case:
  Fetching relationships; "walking the graph"
  - RDB: Intermediate layers on fetch: SQL, JDBC, result-sets, and foreign keys. No context. Over-fetching.
  - ODB: Understands object graph natively. Only sends what it needs, in the format it needs. Relationships can be primed for subsequent retrieval.
- Less reliance on second-level cache => easier to cluster.

Less Code

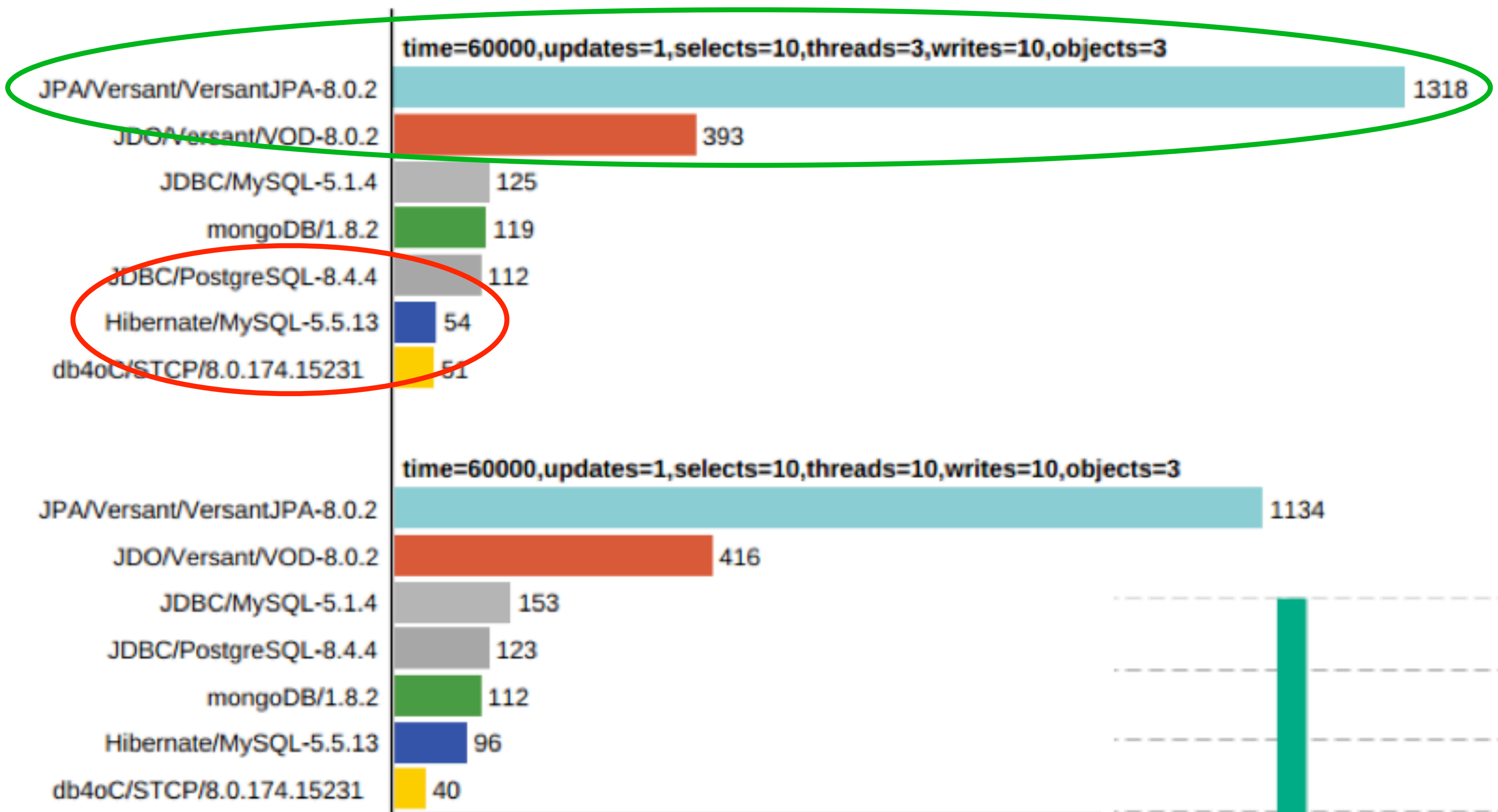
```
@Entity
@Table("player")
@Inheritance(strategy = InheritanceType.JOINED)
public class Player extends AbstractEntity {
    @Column(unique = true, nullable = false,
length = 50, name = "characterName")
    private String name;

    @ManyToOne(
      optional = false, fetch = FetchType.LAZY)
    @JoinColumn(name = "accountTypeId")
    private AccountType type;

    @ManyToMany(mappedBy = "player")
    private Set<Relationship> friends;
    // ...
}
```

```
@Entity
public class Player extends AbstractEntity {
    @Basic(optional=false)
    private String name;

    @ManyToOne(
    optional = false, fetch = FetchType.LAZY)
    private AccountType type;

    @ManyToMany
    private Set<Relationship> friends;
    // ...
}
```

Same Annotations; Same Concepts
- Simple Migration of application code
- Easy to understand for the JPA familiar

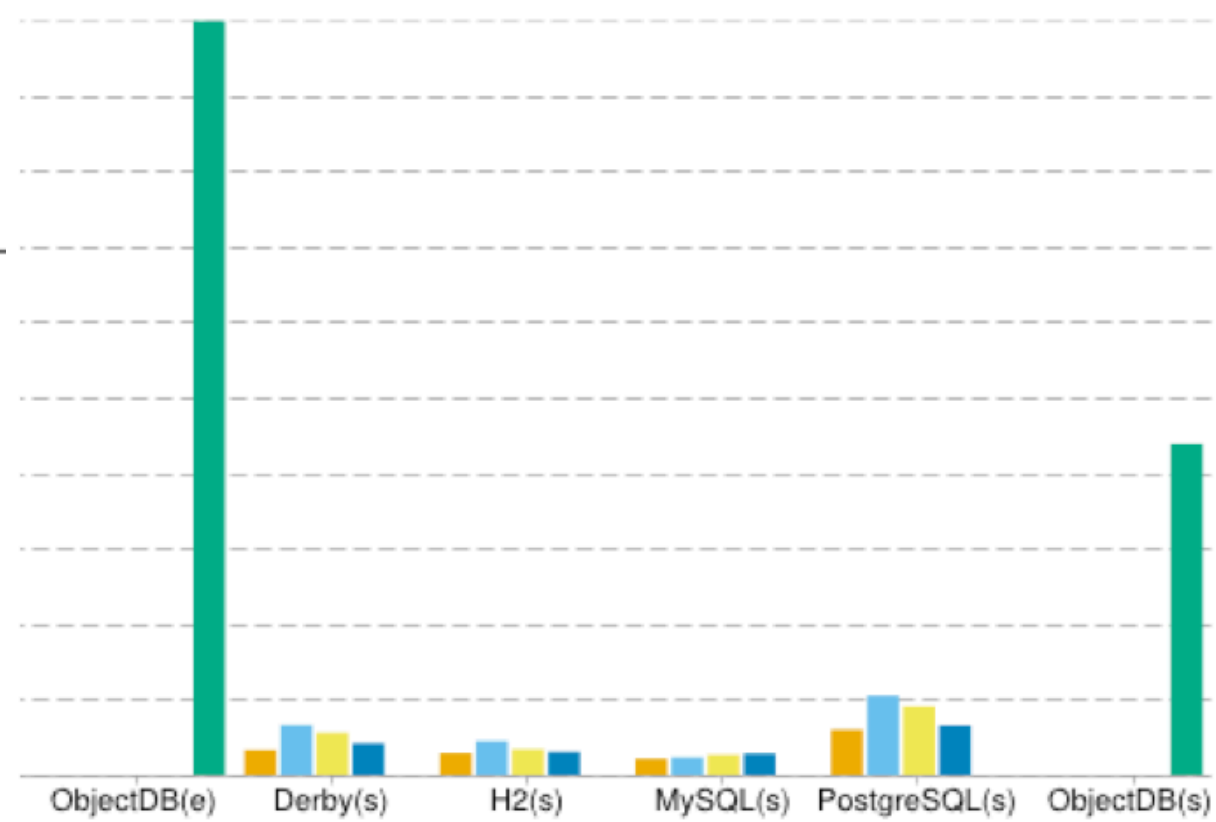
No "Schema Helpers" Required
- Data model matches 1-1 with object
- No confusing "who owns who" questions
- Column names in code feels dirty

Performance



Object DBs are Fast.

Very, Very Fast.



http://polepos.sourceforge.net
"The open source database benchmark"

http://www.jpab.org
"The JPA Benchmark"

Versant Features

- LOIDs for global identity
- Sharded querying
- Selective storage (Object X goes in DB Y)

```
Object x = /*..*/; Database y = /* .. */;
em.persist(y, x);
```

- Transparent, networked object graphs
  (X references Y on different nodes)
- Moving content between nodes via API

```
Object objToMove = em1.get(...);
Database target = /*.. lookup appropriate node ..*/
em2.merge(target, objToMove, MergeMode.ON_CREATE_PRESERVE_LOID);
em1.delete(objToMove); /*.. remove from old DB ..*/
```

A Few More

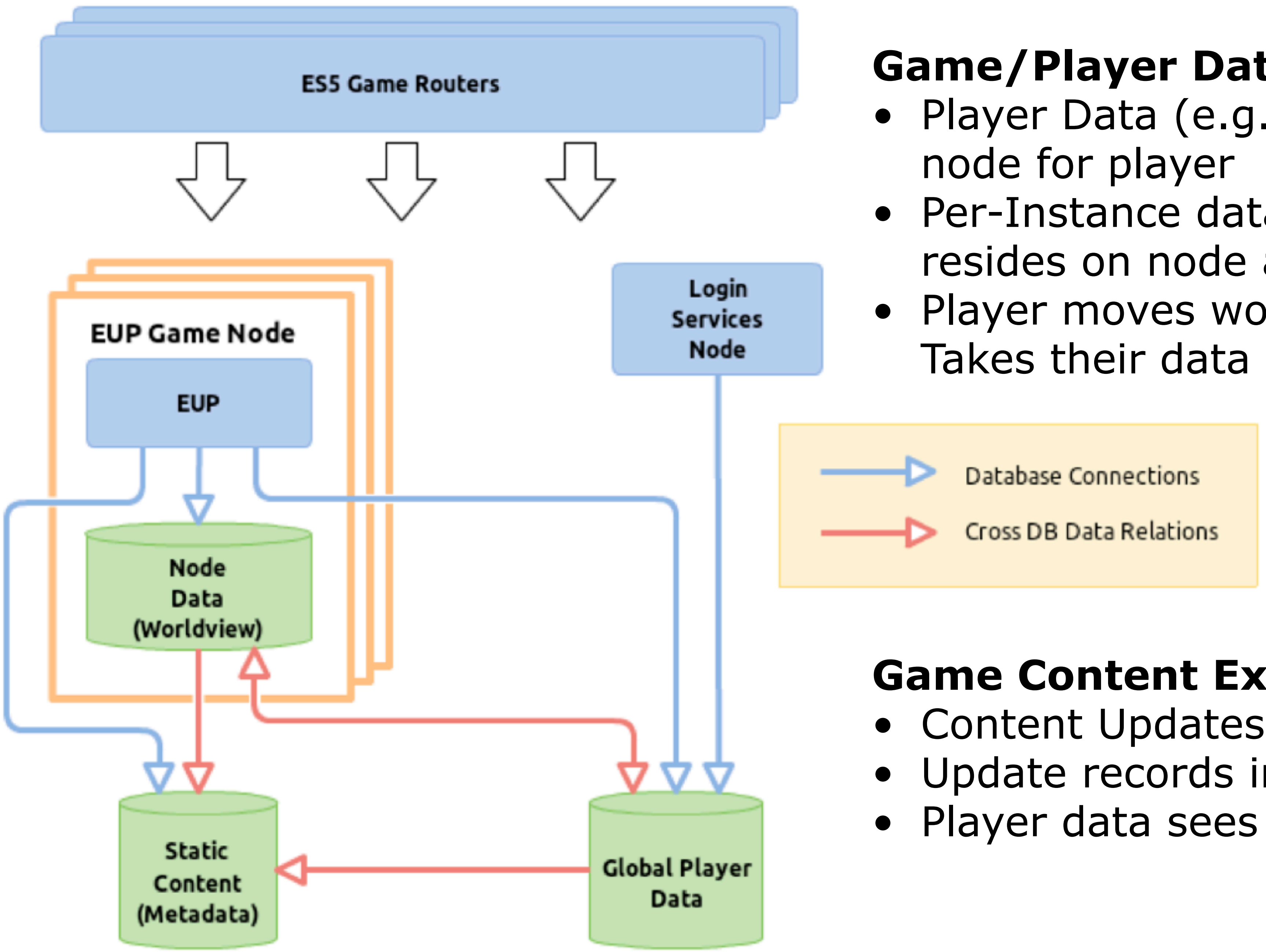
- Schema migration is built-in, intuitive.
- Data evolution in code:

```
@com.electrotank.eup.MarkedForDeletion
private String oldField;
@PostLoad
void onLoad() {
  if(oldField != null) { /* migrate and set null */ }
}
```

- Data migration as a feature

```
Object content = /* the content to migrate */;
Database nextDb = /* the next DB in the server tiers */
em.merge(nextDb,content,MergeMode.ON_CREATE_PRESERVE_LOID);
```

Putting it All Together



Game/Player Data

- Player Data (e.g. Inventory) resides on node for player
- Per-Instance data (e.g. World Items) resides on node as well
- Player moves worlds, changing nodes. Takes their data with them.

Game Content Experience

- Content Updates in test environment
- Update records in Static Content node(s)
- Player data sees update immediately

** Not necessarily 1-1 EUP to DB*

Why This Matters?

- Multiple game nodes allow for true horizontal scalability
- With RDBs, multiple DB nodes is usually impractical (read: very hard). Becomes bottleneck and single point of failure

  (This is an oft-accepted limitation in clustered apps, however)

- Object DBs allow data/DB-traffic balancing along with player simulation balancing
- Per-game configurable scaling model

electrotank®

Thank You! Questions?

R.J. Lorimer - Electrotank, Inc.

E-mail: rjlorimer@electrotank.com
LinkedIn: http://www.linkedin.com/in/rjlorimer
Twitter: @realjenius
Company: http://www.electrotank.com
Personal: http://www.realjenius.com

Robert Greene - Versant Corporation

E-mail: rgreene@versant.com