Building Highly Available Multi-Tier Applications

Emmanuel Cecchet
Chief architect - Continuent

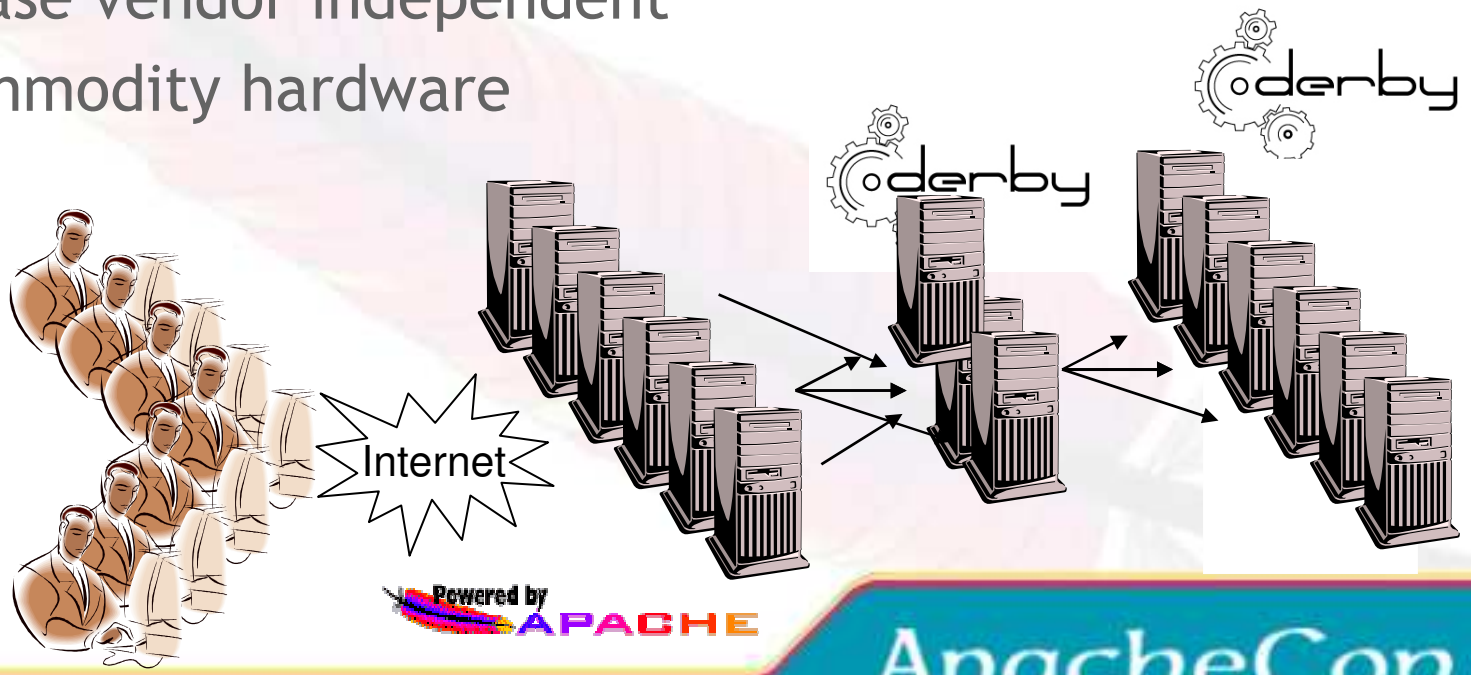
continuent
Open. Always Available.

http://www.continuent.org

ApacheCon
Europe 06

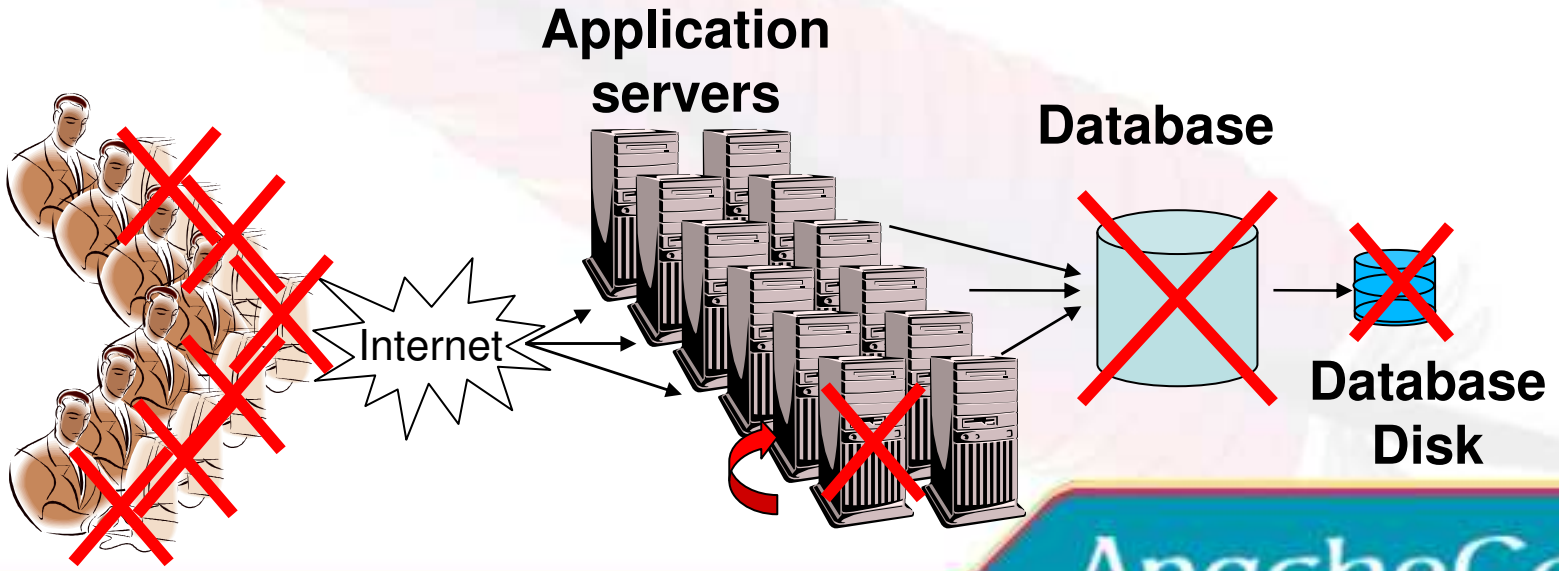
Motivations

- Database tier should be
  - scalable
  - highly available
  - without modifying the client application
  - database vendor independent
  - on commodity hardware



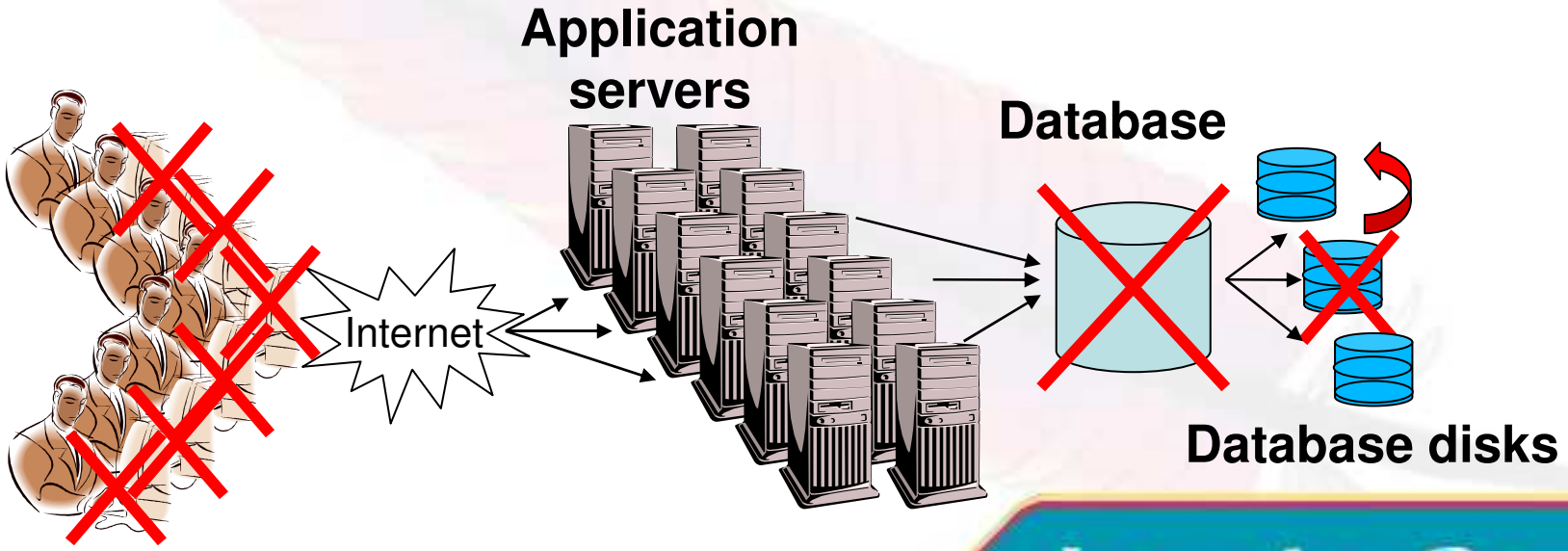
Single database issue

- Clients connect to the application server
- Application server builds web pages with data coming from the database
- Application server clustering solves application server failure
- Database outage causes overall system outage



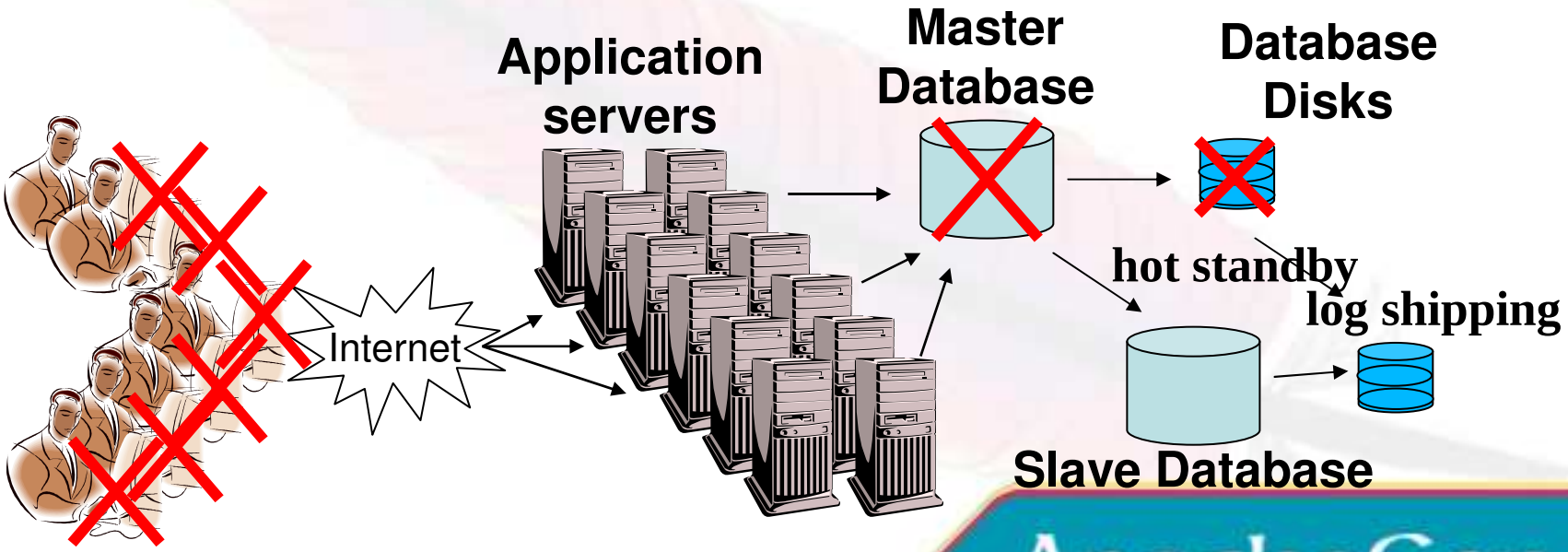
Disk replication/clustering

- Eliminates the single point of failure (SPOF) on the disk
- Disk failure does not cause database outage
- Database outage problem still not solved



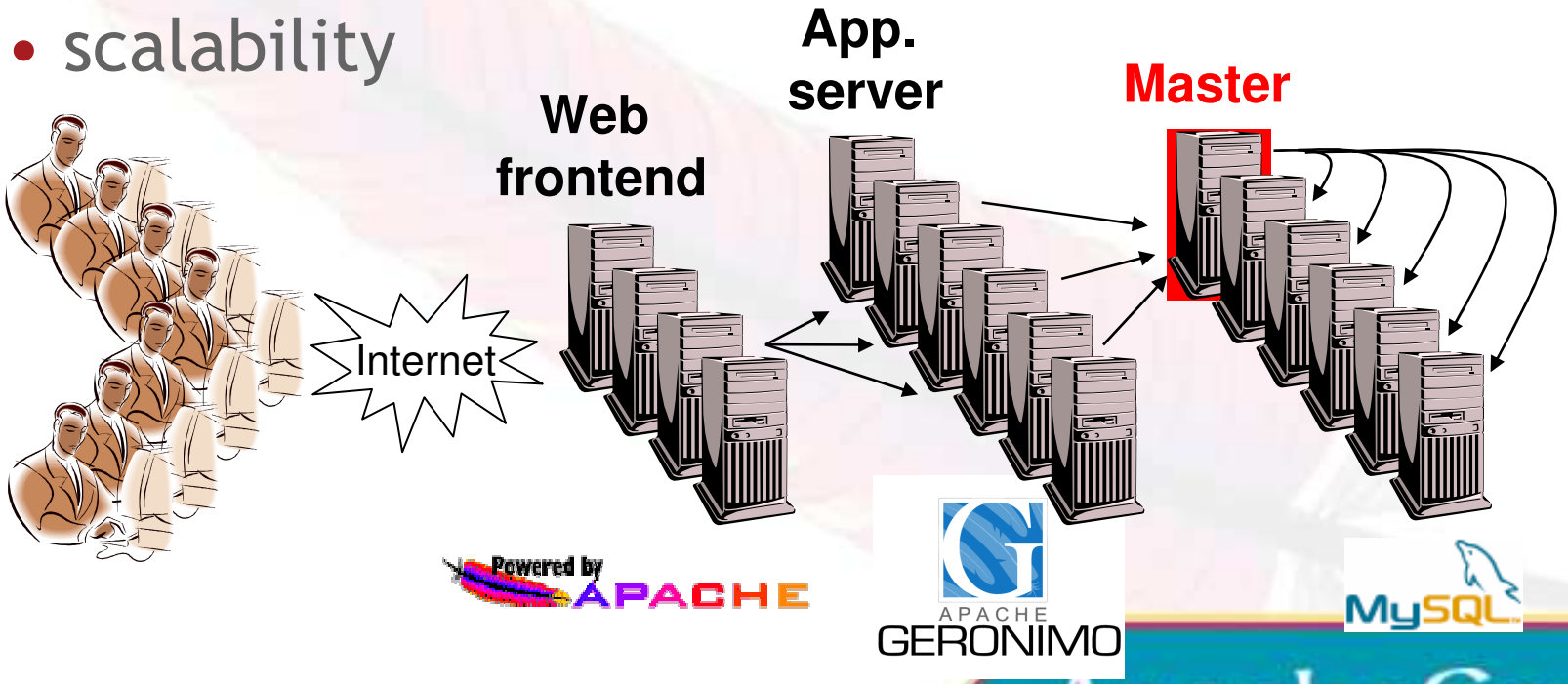
Master/slave replication

- Lazy replication at the disk or database level
- No scalability
- Data lost at failure time
- System outage during failover to slave
- Failover can take several hours



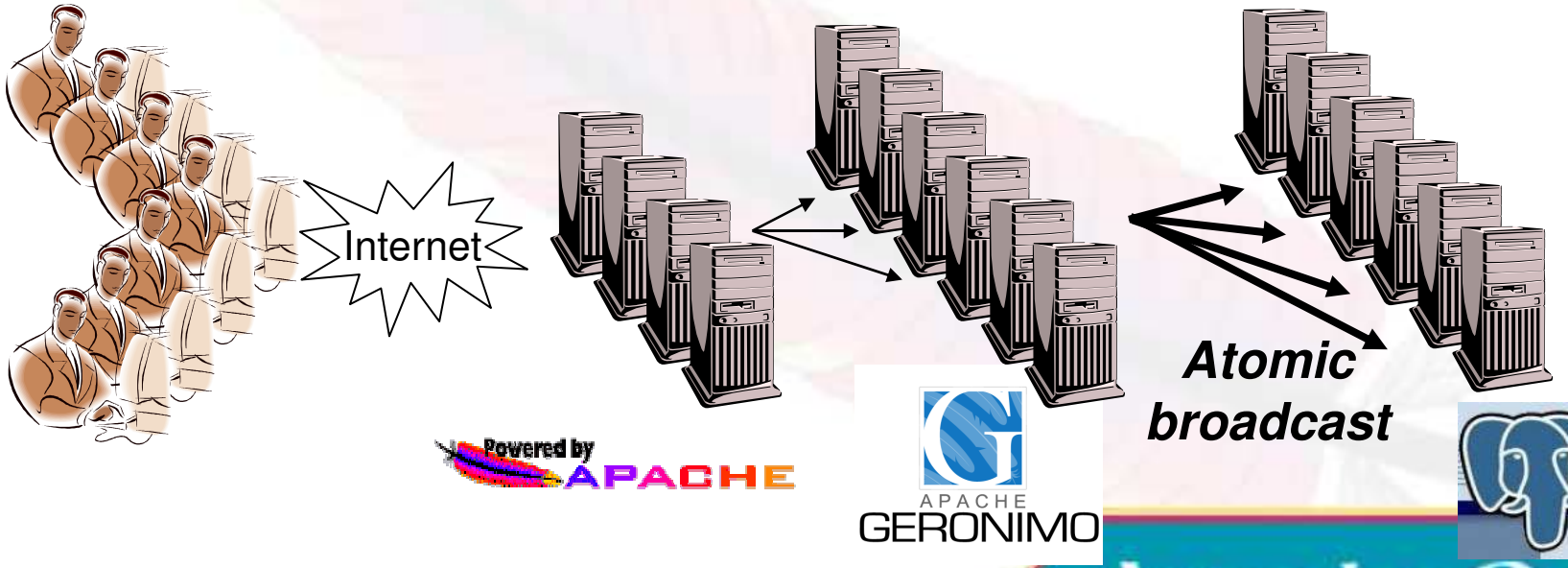
Scaling the database tier
Master-slave replication

- Cons
  - failover time/data loss on master failure
  - read inconsistencies
  - scalability



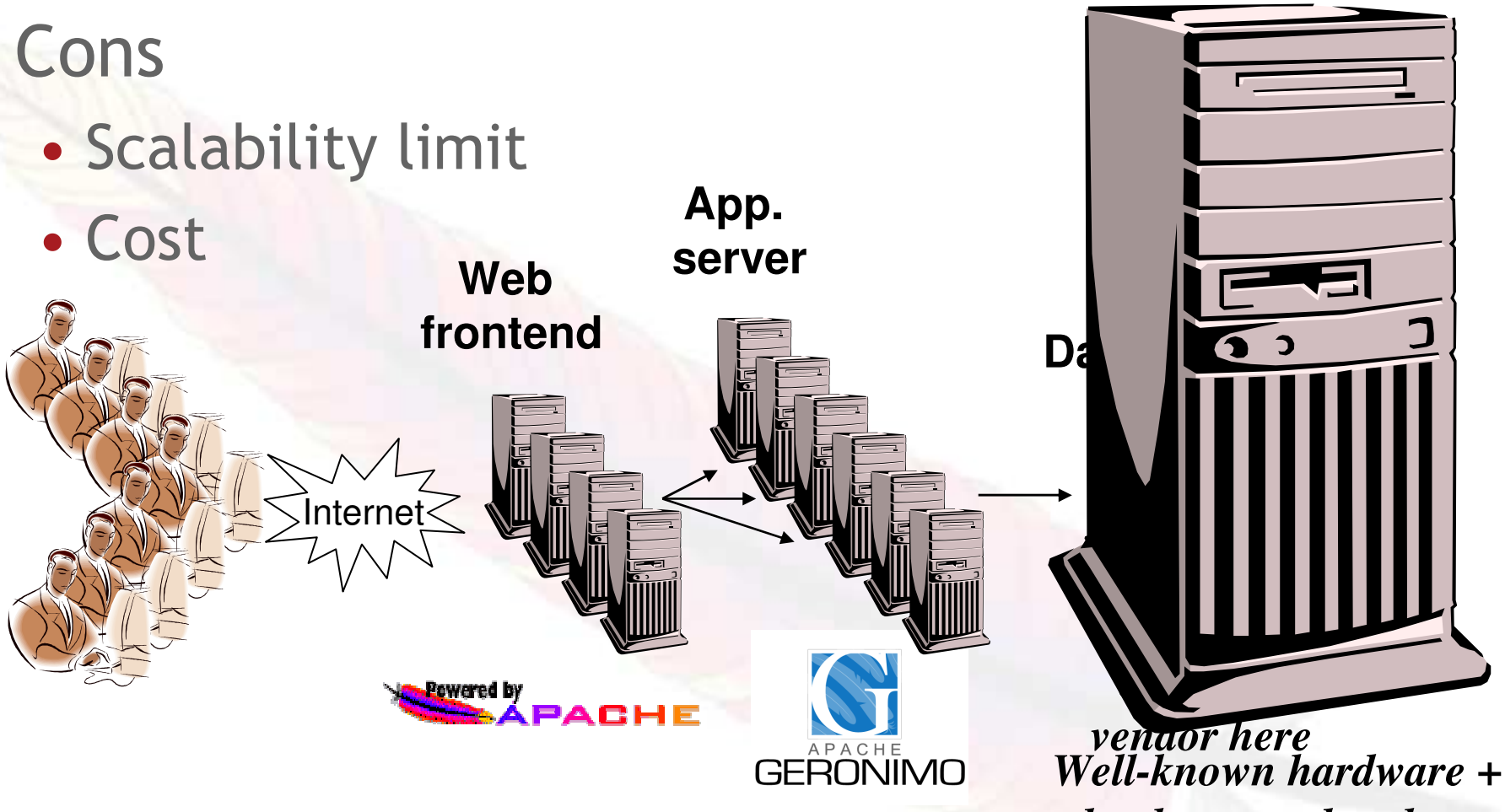
Scaling the database tier
Atomic broadcast

- Cons
  - atomic broadcast scalability
  - no client side load balancing
  - heavy modifications of the database engine



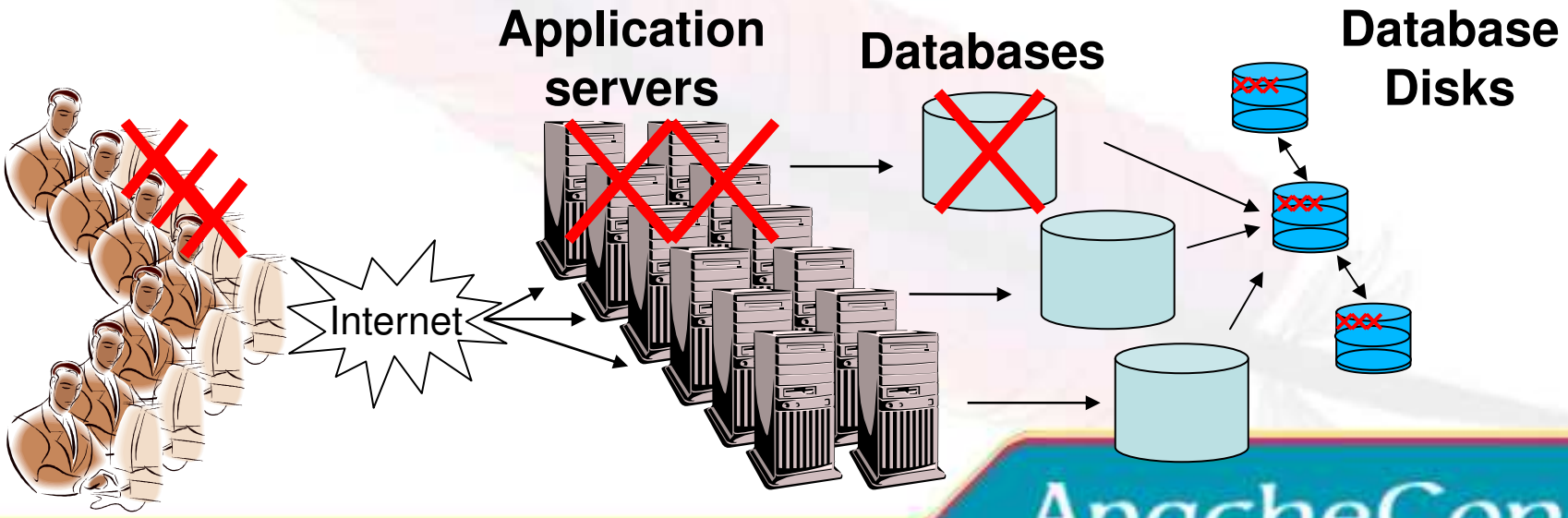
Scaling the database tier – SMP

- Cons
  - Scalability limit
  - Cost



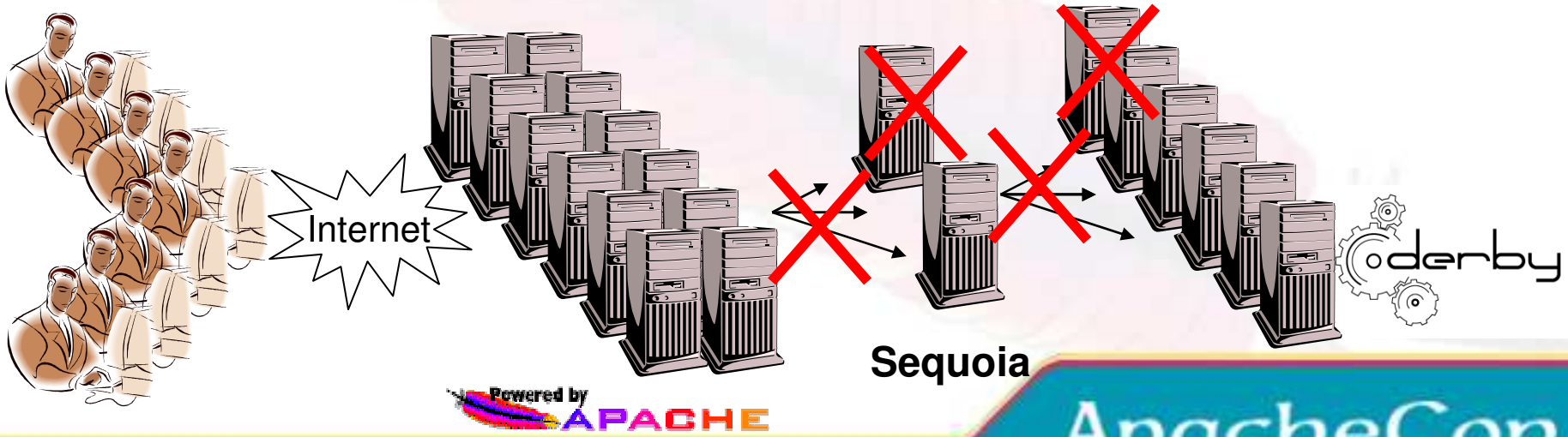
Database clustering with shared disk

- Multiple database instances share the same disk
- Disk can be replicated to prevent SPOF on disk
- No dynamic load balancing
- Database failure not transparent to users (partial outage)
- Manual failover + manual cleanup needed



Transparent failover

- Failures can happen
  - in any component
  - at any time of a request execution
  - in any context (transactional, autocommit)
- Transparent failover
  - masks all failures at any time to the client
  - perform automatic retry and preservers consistency



Sequoia competitive advantages

- High availability with fully transparent failover
- Online upgrade and maintenance operations
- Scalability with dynamic load balancing
- Commodity hardware
- No modification to existing client applications
- Database vendor independent
- Heterogeneity support
- Platform independent

Outline

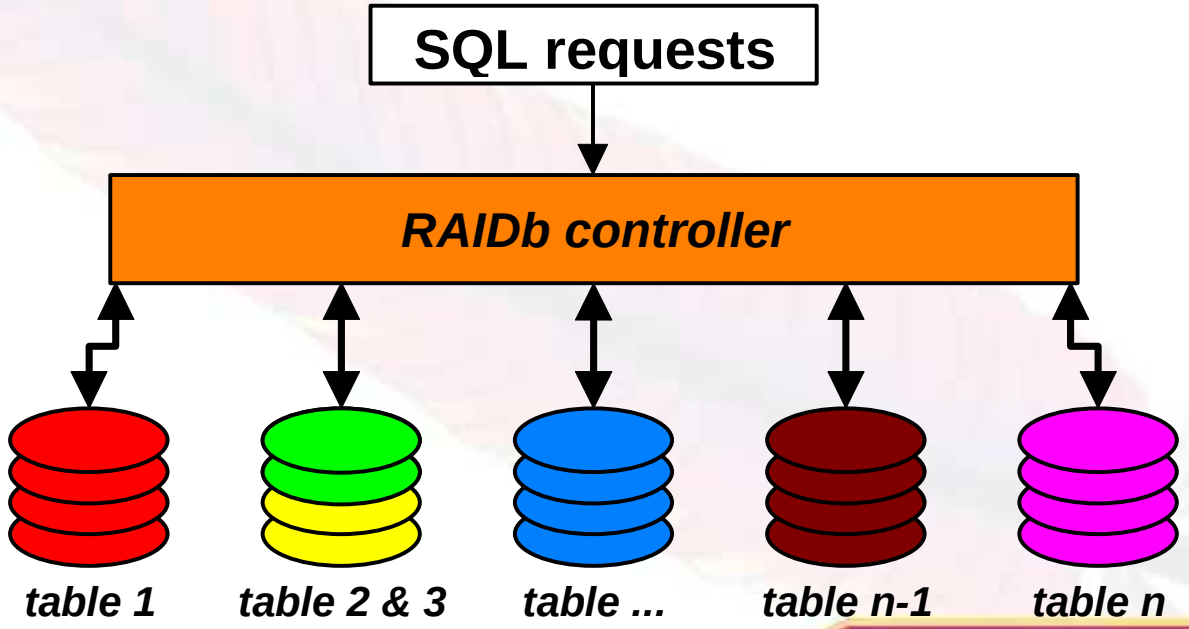
- RAIDb
- Sequoia
- Building an HA solution
- Scalability
- High availability

RAIDb concept

- Redundant Array of Inexpensive Databases
- RAIDb controller
  - gives the view of a single database to the client
  - balance the load on the database backends
- RAIDb levels offers various tradeoff of performance and fault tolerance

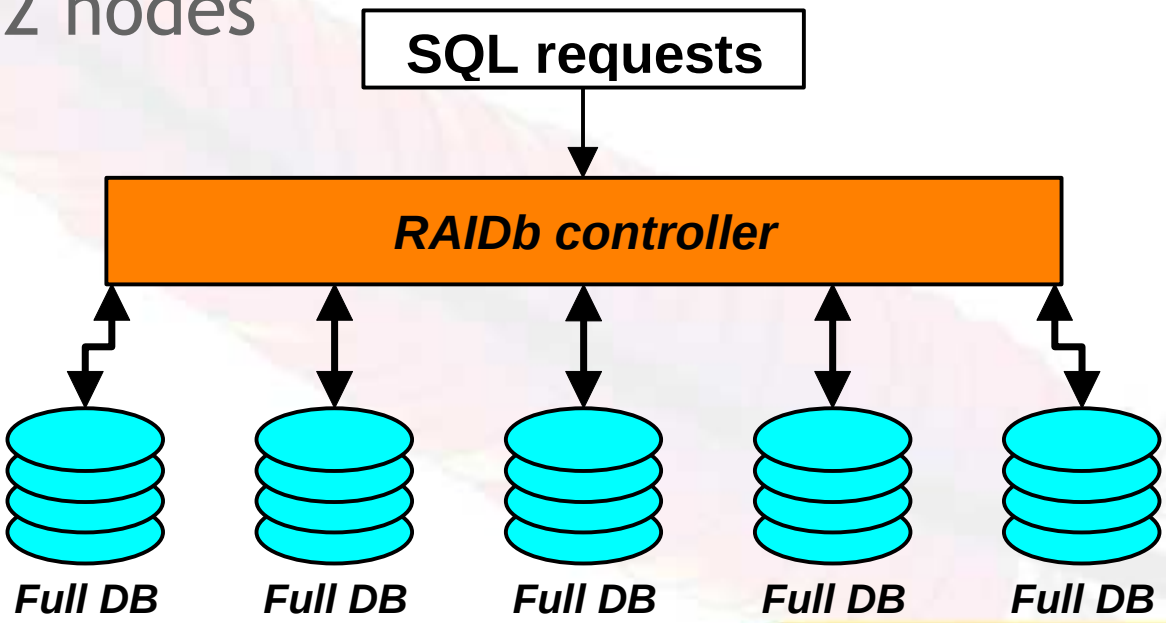
RAIDb levels

- RAIDb-0
  - partitioning
  - no duplication and no fault tolerance
  - at least 2 nodes



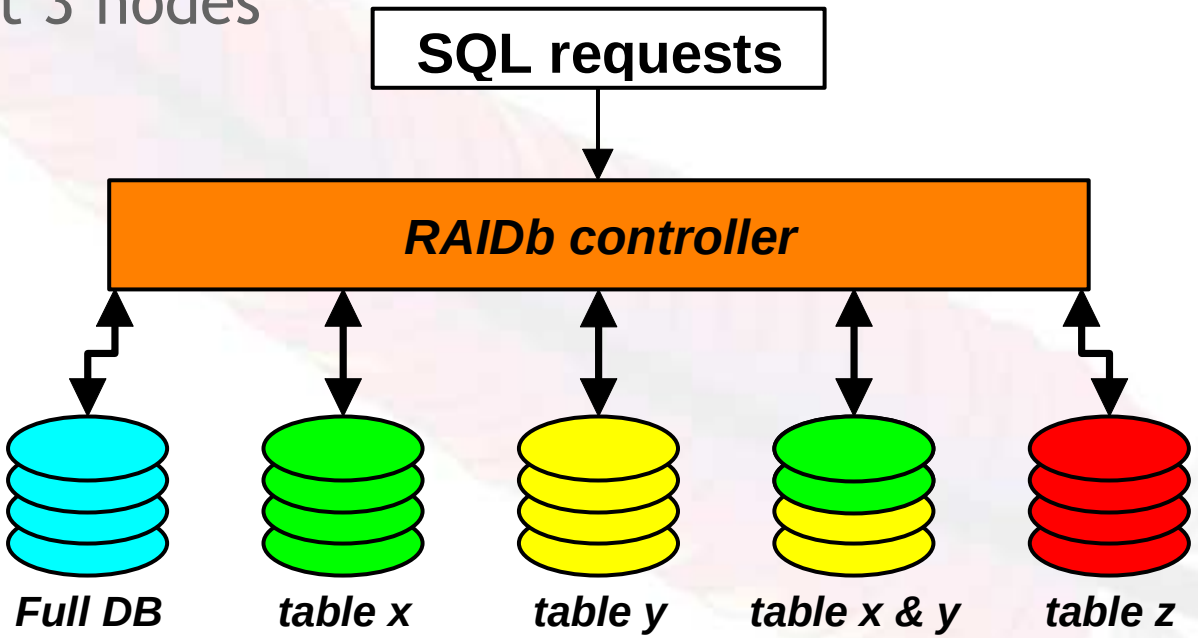
RAIDb levels

- RAIDb-1
  - mirroring
  - performance bounded by write broadcast
  - at least 2 nodes

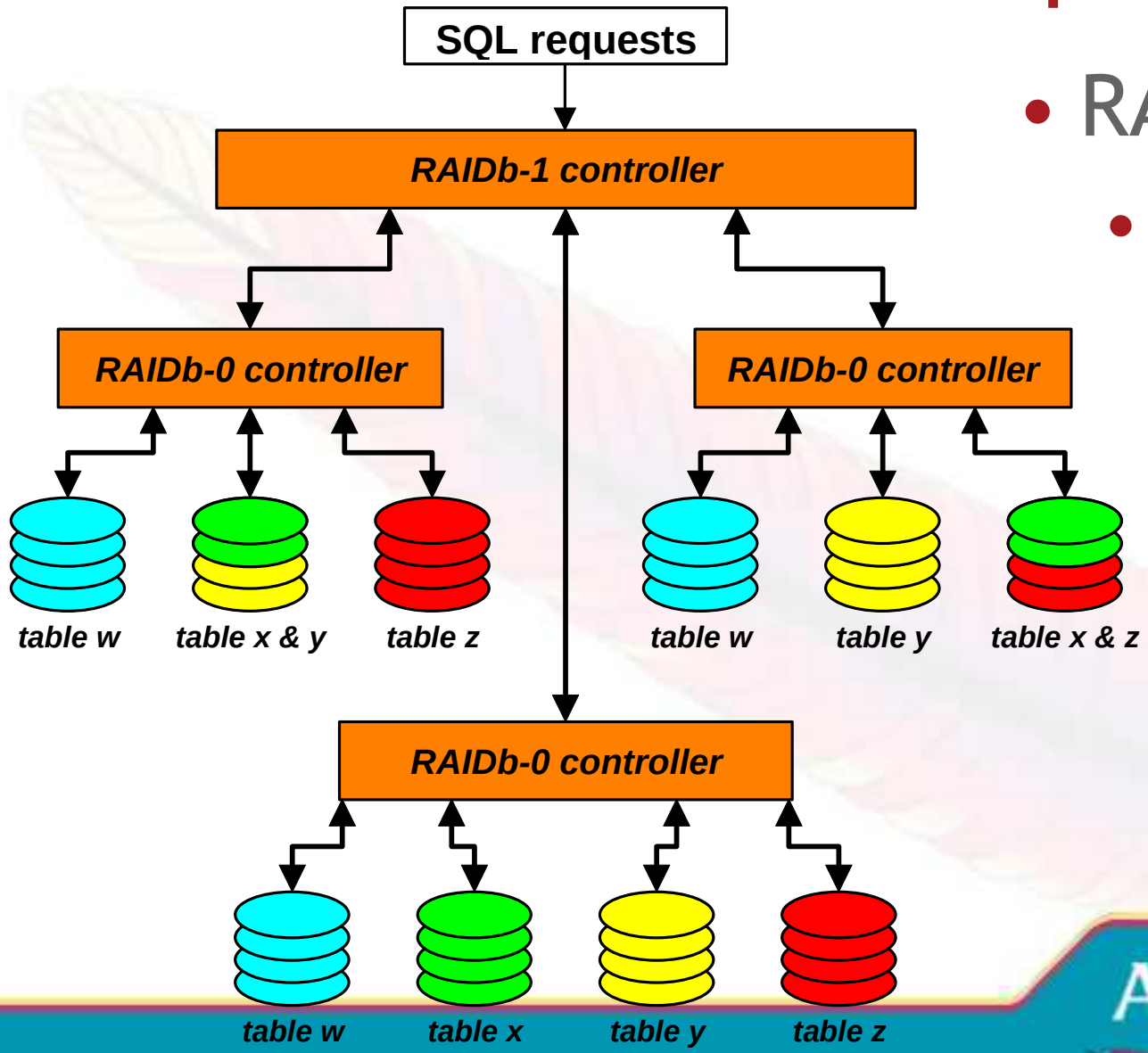


RAIDb levels

- RAIDb-2
  - partial replication
  - at least 2 copies of each table for fault tolerance
  - at least 3 nodes



RAIDb levels composition



- RAIDb-1-0
  - no limit to the composition deepness

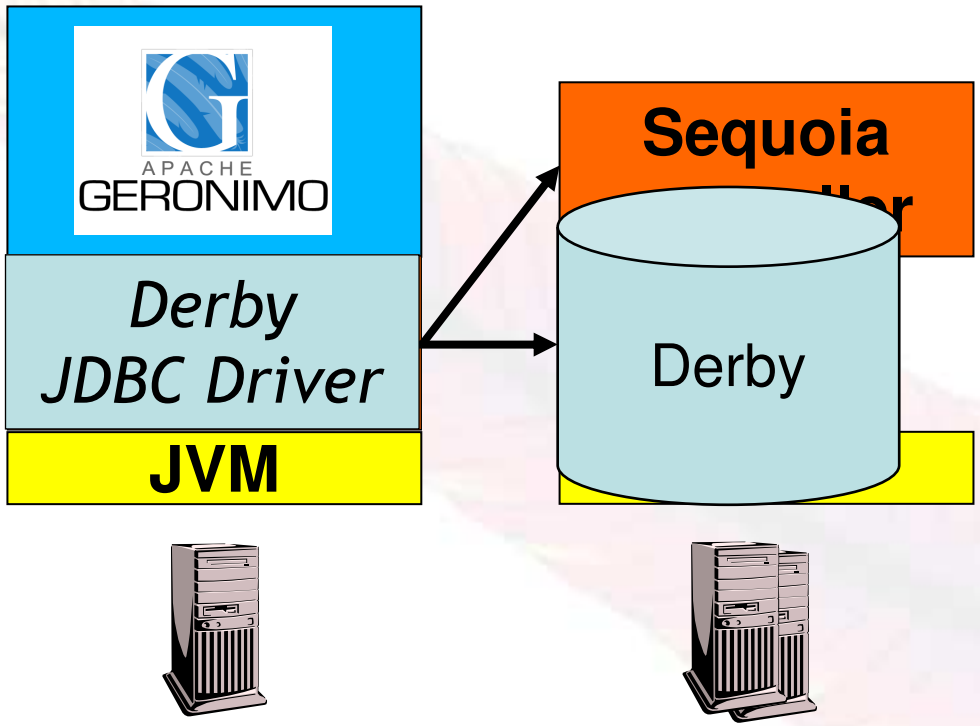
Outline

- RAIDb
- Sequoia
- Geronimo/Sequoia/Derby
- Scalability
- High availability

Sequoia overview

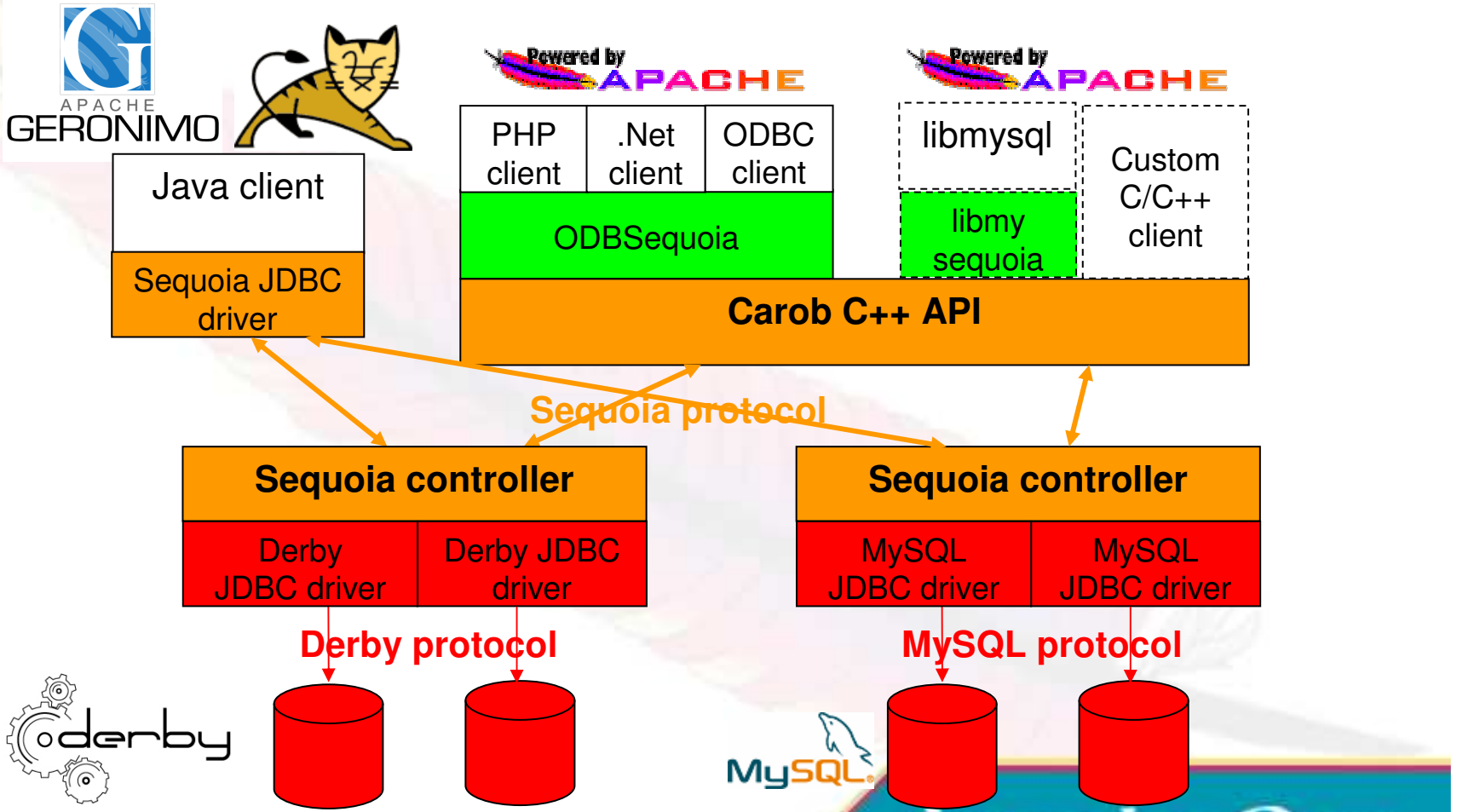
- Middleware implementing RAIDb
  - 100% Java implementation
  - open source (Apache v2 License)
- Two components
  - Sequoia driver (JDBC, ODBC, native lib)
  - Sequoia Controller
- HA and load balancing features
- Database neutral (generic ANSI SQL)

Sequoia architectural overview



Sequoia drivers (1/2)

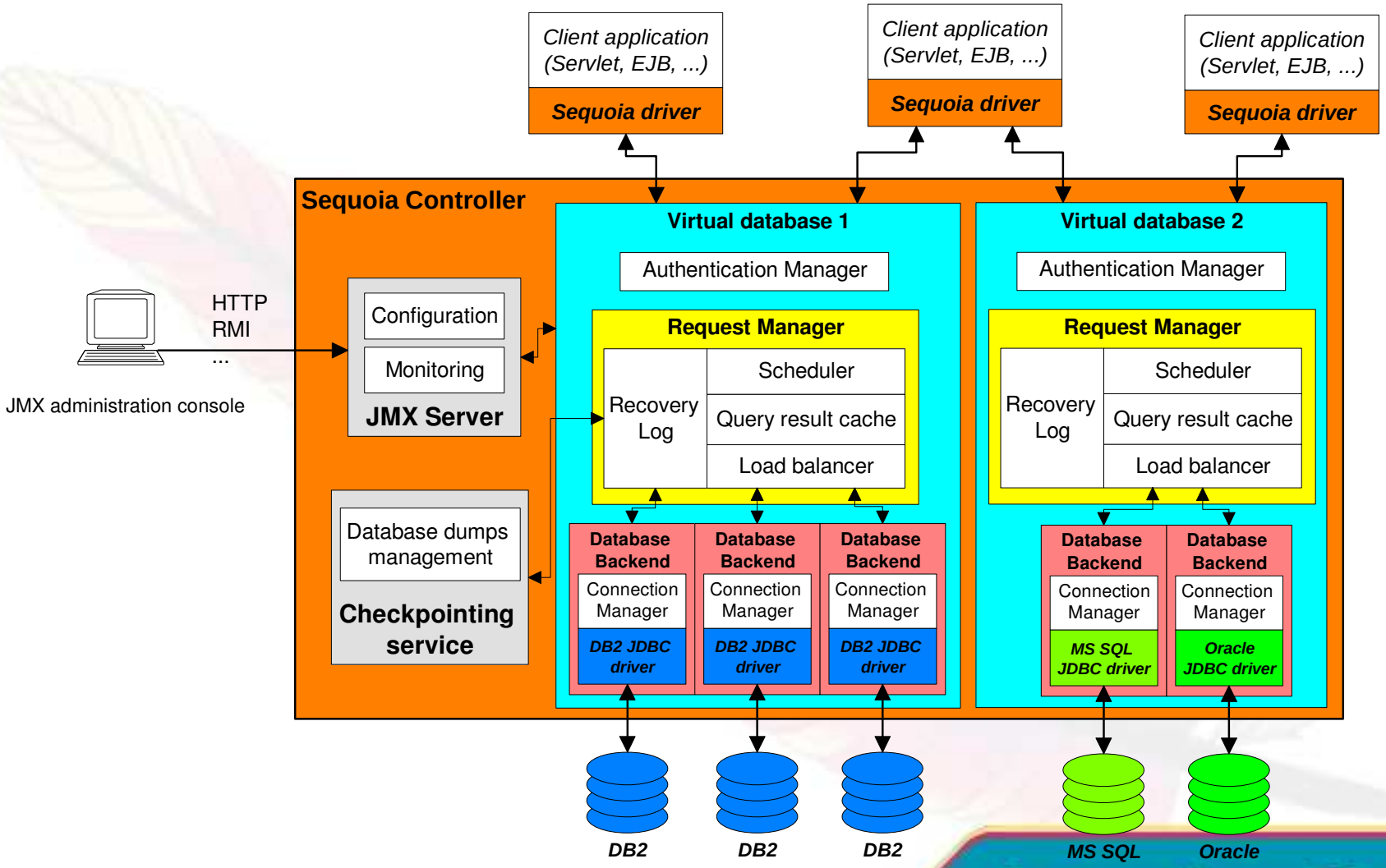
- Visit http://carob.continuent.org



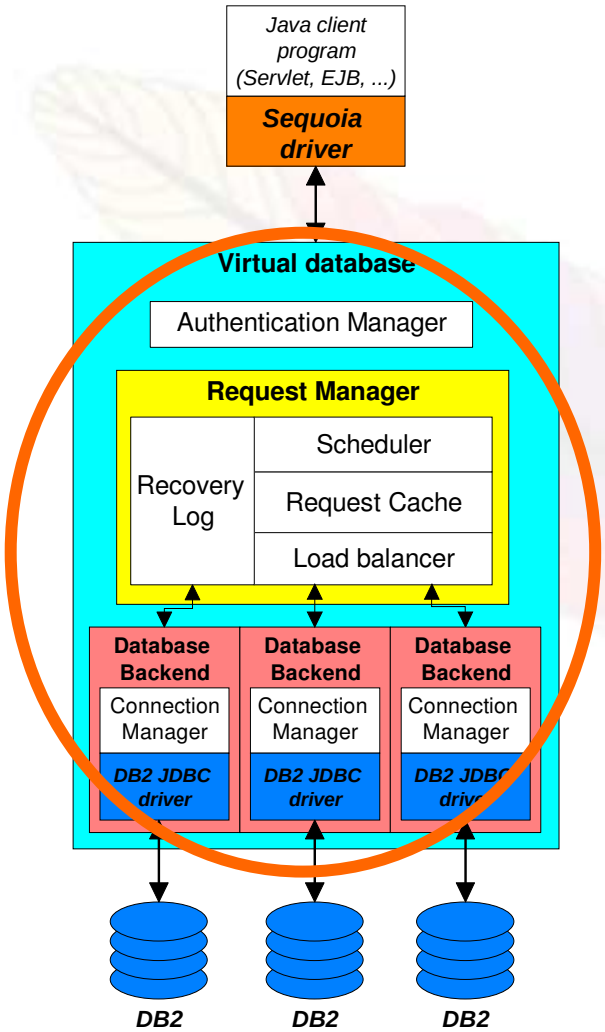
Sequoia drivers (2/2)

- Provide load balancing between controllers for connection establishment
- Controller failure detection
- Transparent failover for any request in any context (including metadata) on controller failure
- Transparent SQL proxying
- Optional SSL if needed

Inside the Sequoia Controller

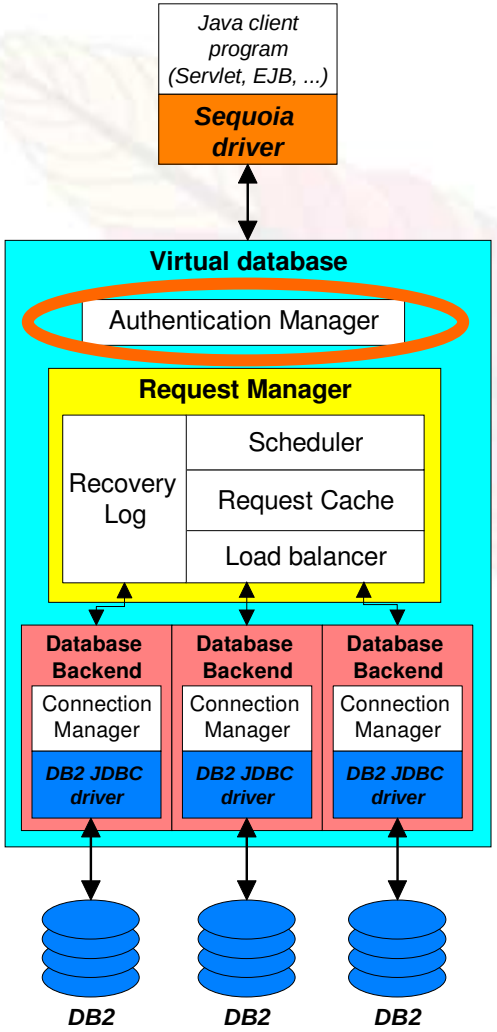


Virtual Database



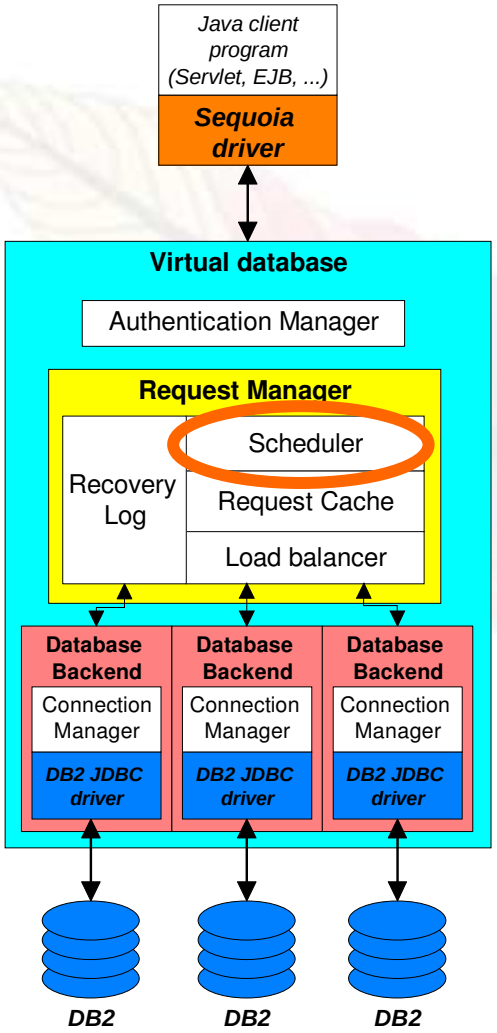
- Gives the view of a single database
- Establishes the mapping between the database name used by the application and the backend specific settings
- Backends can be added and removed dynamically
- Backends can be transferred between controllers
- configured using an XML configuration file

Authentication Manager



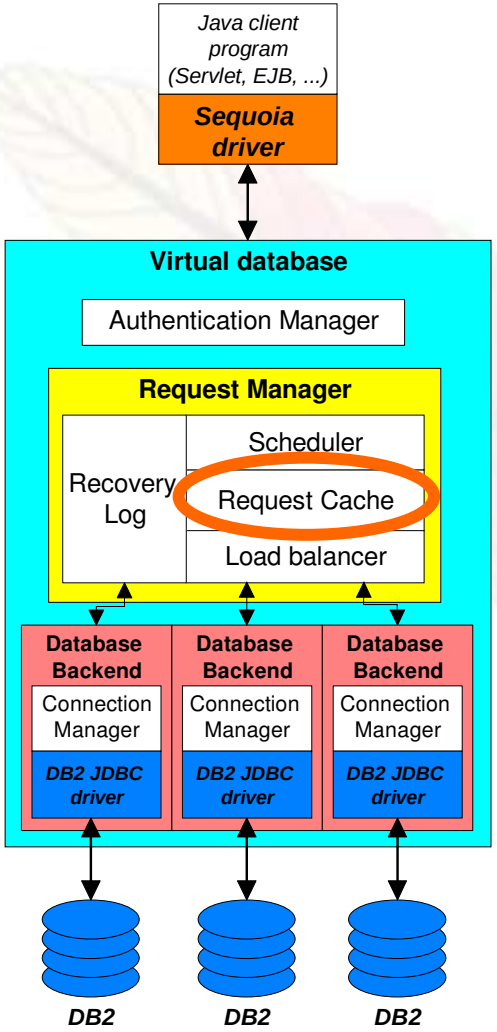
- Matches real login/password used by the application with backend specific login/ password
- Administrator login to manage the virtual database
- Transparent login proxying available

Scheduler



- Manages concurrency control
- Provides support for cluster-wide checkpoints
- Assigns unique ids to requests and transactions
- Ensure serializable transactional order
- Relies on database (row-level) locking

Request cache



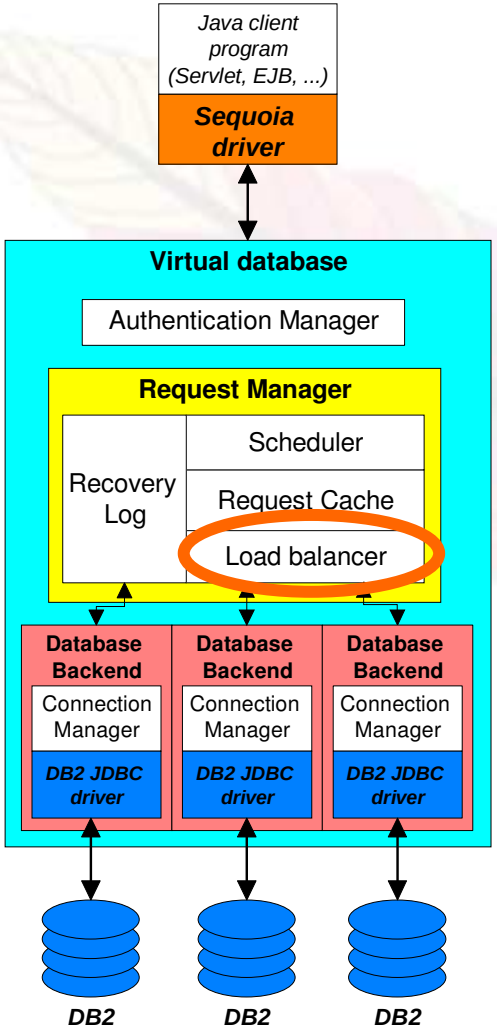
- 3 optional caches
  - tunable sizes
- parsing cache
  - parse request skeleton only once
  - INSERT INTO t VALUES (?,?,?)
- metadata cache
  - column metadata
  - fields of a request
- result cache
  - caches results from SQL requests
  - tunable consistency
  - optimizations for findByPk requests

Result cache

- Cache contains a list of SQL→ResultSet
- Policy defined by queryPattern→Policy
- 3 policies
  - EagerCaching: variable granularities for invalidations
  - RelaxedCaching: invalidations based on timeout
  - NoCaching: never cached

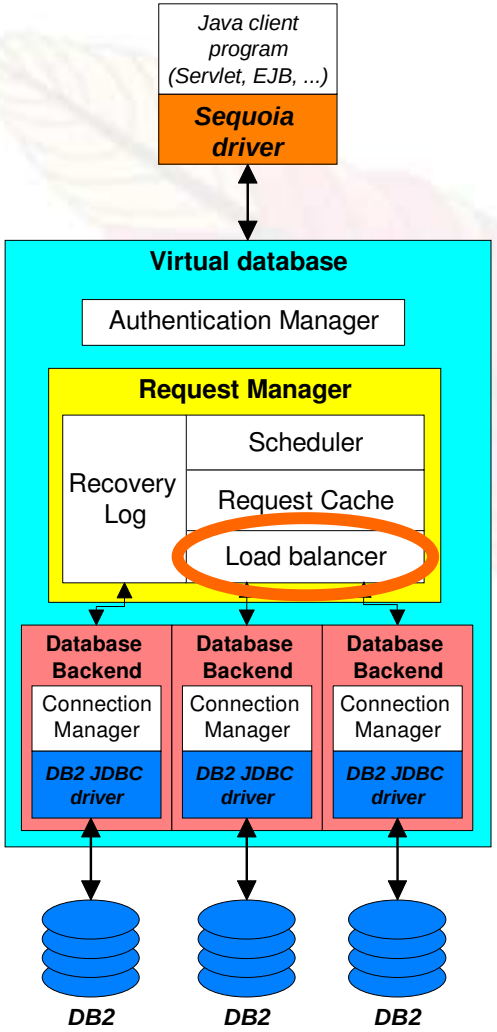
RUBiS bidding mix with 450 clients	No cache	Coherent cache	Relaxed cache
Throughput (rq/min)	3892	4184	4215
Avg response time	801 ms	284 ms	134 ms
Database CPU load	100%	85%	20%
Sequoia CPU load	-	15%	7%

Load balancer 1/2



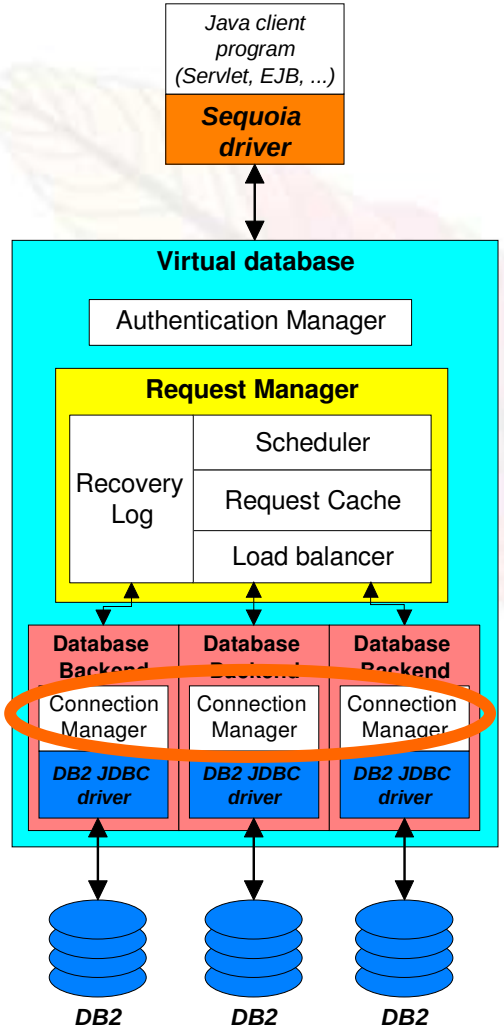
- RAIDb-0
  - query directed to the backend having the needed tables
- RAIDb-1
  - read executed by current thread
  - write multicast through group communication and executed in parallel
  - asynchronous execution possible
  - if one node fails but others succeed, failing node is disabled
- RAIDb-2
  - same as RAIDb-1 except that writes are sent only to nodes owning the updated table

Load balancer 2/2



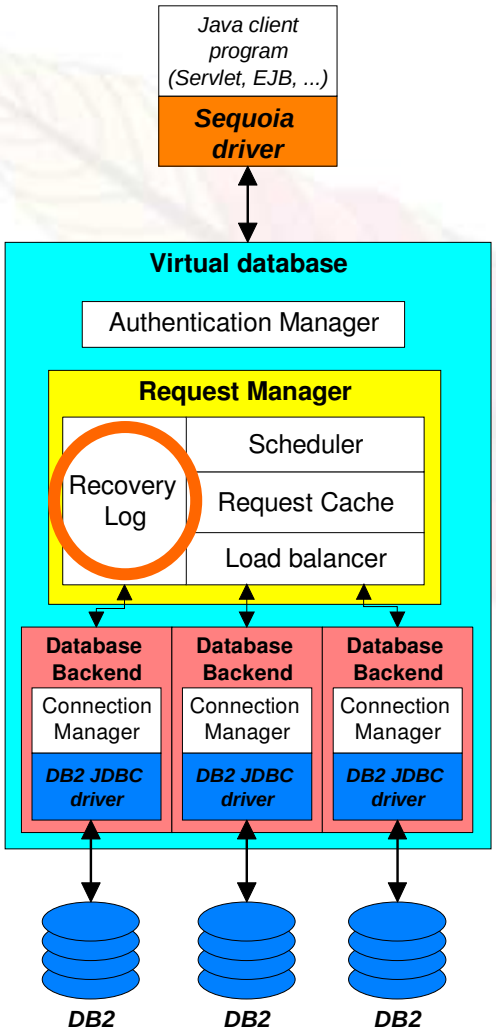
- Static load balancing policies
  - Round-Robin (RR)
  - Weighted Round-Robin (WRR)
- Least Pending Requests First (LPRF)
  - request sent to the node that has the shortest pending request queue
  - efficient even if backends do not have homogeneous performance

Connection Manager



- Sequoia JDBC driver provides transparent connection pooling
- Connection pooling for a backend
  - no pooling
  - blocking pool
  - non-blocking pool
  - dynamic pool
- Connection pools defined on a per login basis
  - resource management per login
  - dedicated connections for admin

Recovery Log



- Transactional log for recovery/resync
- Checkpoints are associated with database dumps
- Record all updates and transaction markers since a checkpoint
- Store log information in a database

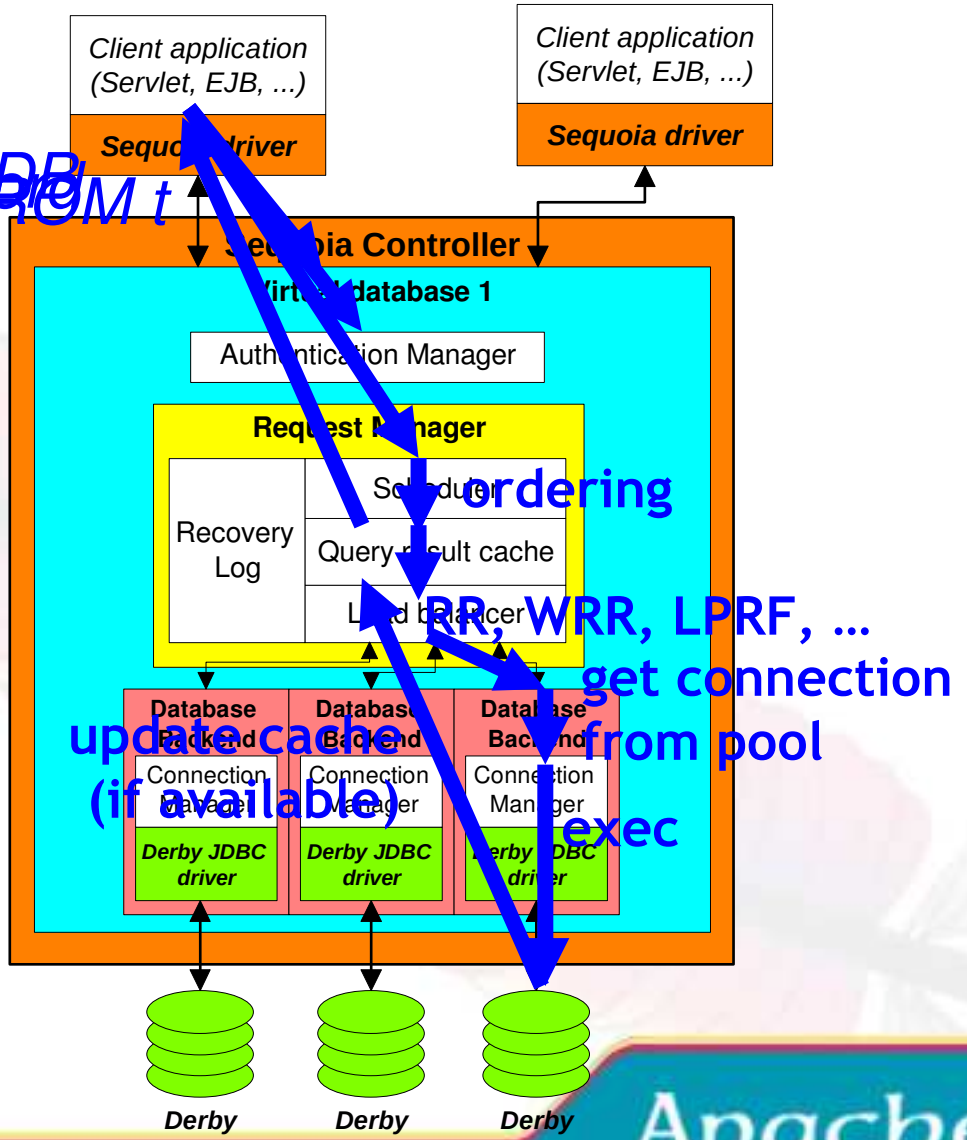
Functional overview

- Connection establishment
- Read request
- Write request
- Failure handling
- Recovery
- Summary

Connection establishment

- Sequoia JDBC URL
  - jdbc:sequoia://controller1[:port1],controller2[:port2]/vdbname
- Driver connection to a controller according to a policy defined in the URL (random, round-robin, ordered, …)
- Controller connections to a physical database use a connection pool defined per backend
- All queries on a driver connection go to the same controller but controller load balance queries on the underlying backends

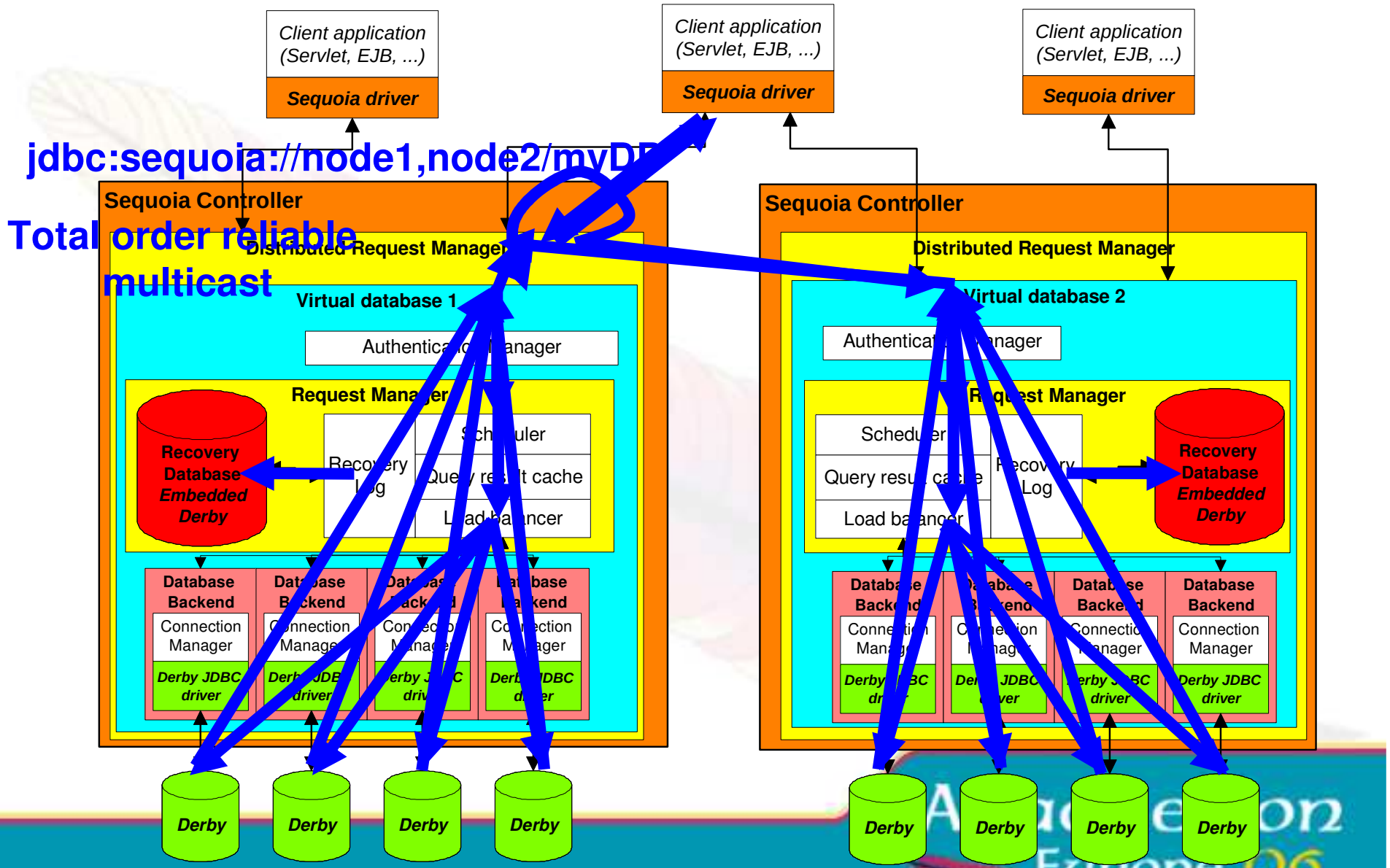
Sequoia read request



Write request

- Query broadcast using total order between controllers
- Backends execute query in parallel (possibly asynchronously)
- Automatically disable backends that fail if others succeed
- Update recovery log (transactional log) asynchronously
- Non-conflicting transactions execute in parallel
  - table-based locking used as a hint for detecting conflicts
  - conflicting queries execute in a single-threaded queue
  - stored procedures execute one at a time

Controller replication



Sequoia failure handling (1/2)

- Failure of a connection to a controller
  - transparently reconnected to another controller (according to user defined policy)
  - failure inside a transaction restores the transactional context upon reconnection
- backend failure
  - backend removed from load balancer (no noticeable failover time)
  - failure during read: retry on another backend
  - failure during write: ignored if successful on other backends

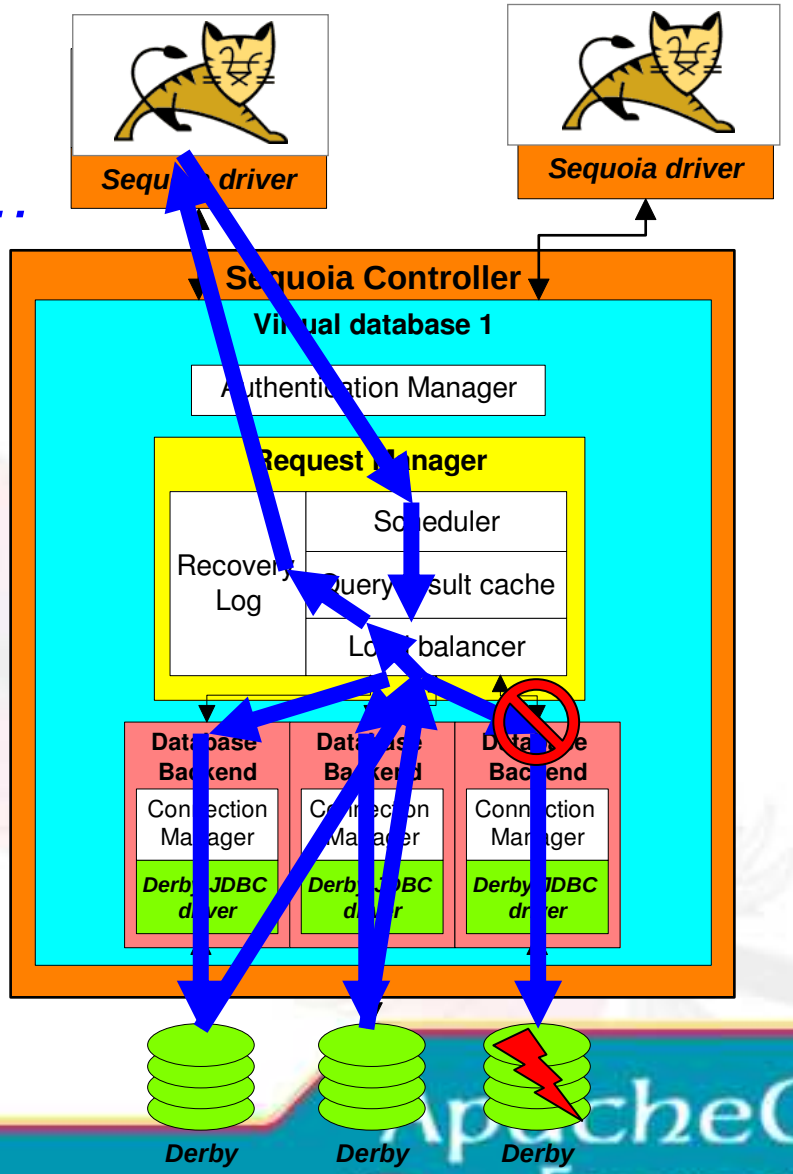
Sequoia failure handling (2/2)

- controller failure
  - database backends attached to dead controller are lost
  - clients are transparently reconnected to another controller (according to user defined policy)
  - failure during read: transparently retried (even within a transaction)
  - failure during write: retrieve from remaining controller(s) failover cache

Node failure with Sequoia

execute *INSERT INTO t …*

- No 2 phase-commit
  - parallel transactions
  - failed nodes are automatically disabled

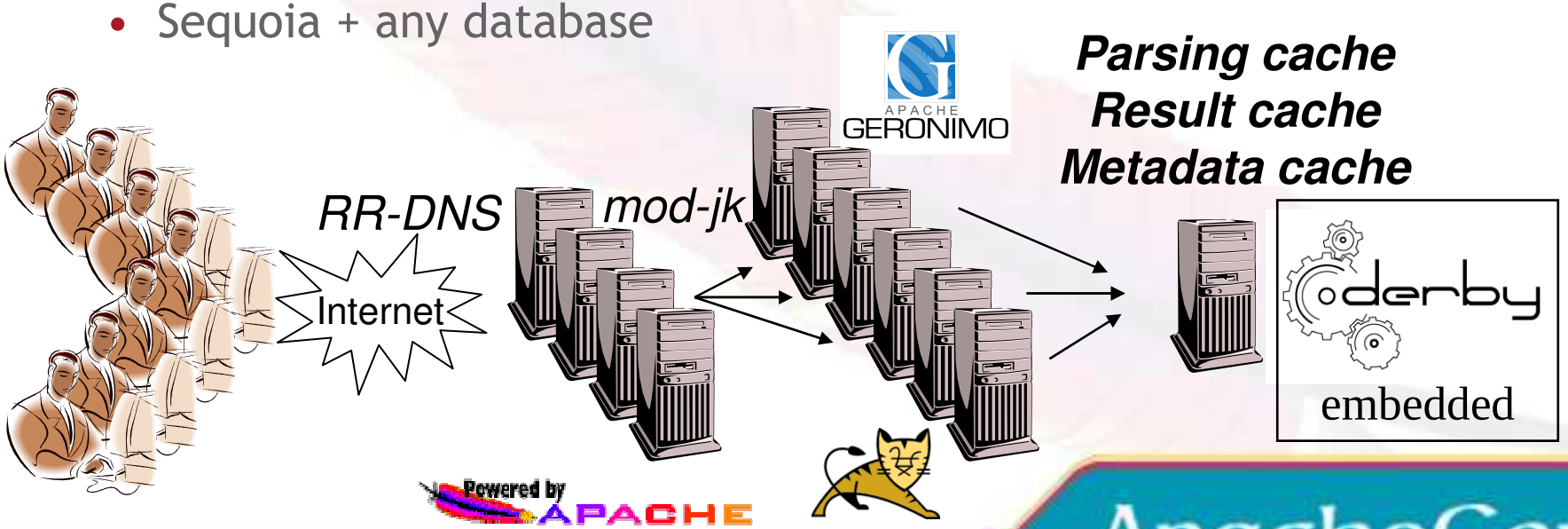


Outline

- RAIDb
- Sequoia
- Building an HA solution
- Scalability
- High availability

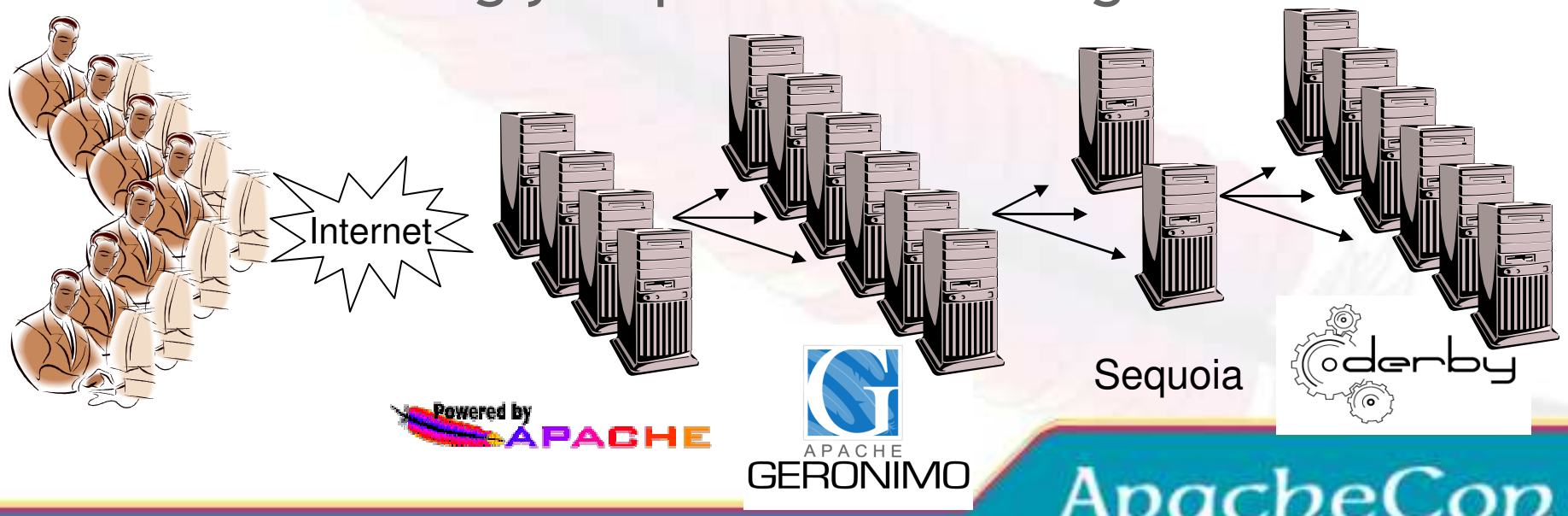
Highly available web site

- Apache clustering
  - L4 switch, RR-DNS, One-IP techniques, LVS, …
- Web tier clustering
  - mod_jk (T4), mod_proxy/mod_rewrite (T5), session replication
- Database clustering
  - Sequoia + any database



Highly available web applications

- Consider MTBF (Mean time between failure) of every hardware and software component
- Take MTTR (Mean Time To Repair) into account to prevent long outages
- Tune accordingly to prevent trashing



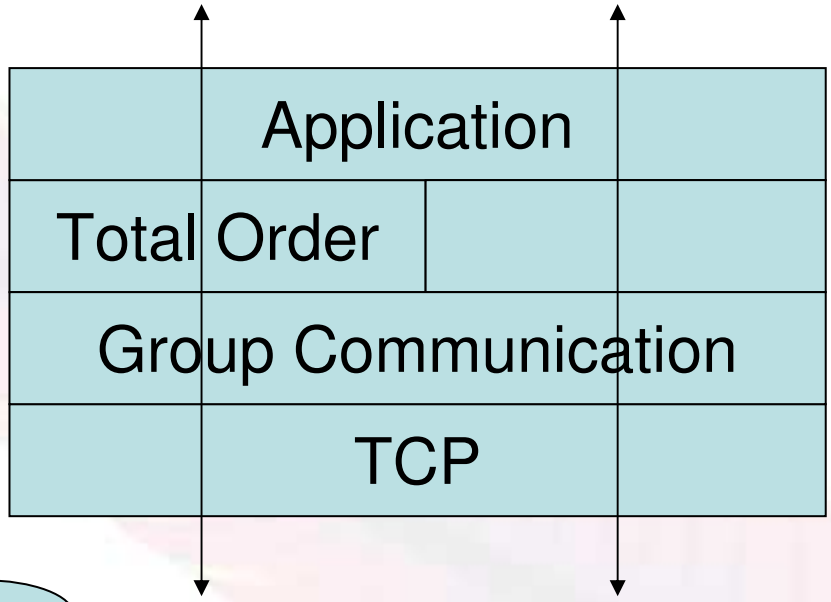
Group communication

- Group communication is the base for replication
- Different tiers might have different needs
  - Ordering (FIFO, Total, causal, …)
  - Delivery (Best effort, reliable, …)
  - Group Membership (View synchrony, …)
  - Network protocols (UDP, TCP, …)
  - LAN vs WAN

Appia

- Flexible group communication library available under APL v2 (http://appia.continuent.org)
- Composed by
  - A **core** used to compose protocols
  - A **set of protocols**
    - group communication, ordering guarantees, atomic broadcast, among other properties
- Created and supported by the Appia team
  - University of Lisbon, Portugal
    - DIALNP research group

Appia Protocol Composition (example)



Atomic Broadcast Channel

Group Communication Channel

Appia features (1/2)

- Possible transport protocols
  - UDP / IP Multicast
  - TCP
  - TCP+SSL
- Existing protocols
  - Causal order
  - Total order
    - Sequencer-based;
    - Token-based;
    - Total-Causal;
    - Hybrid (Adaptive Sequencer/Causal);
    - Optimistic (sequencer-based) Uniform Total Order

Appia features (2/2)

- Event-based delivery model
  - Support for asynchronous events
- Optional view synchrony with failure detector
- Support for QoS constraints

Sequoia/Java configuration

- Keep all original database specific settings as is (SQL dialect, connection test statements, …)
- Copy sequoia-driver.jar in classpath
- Set driver class name to
  org.continuent.sequoia.driver.Driver
- Set JDBC URL to
  jdbc:sequoia://host1,host2/mydb

Sequoia/PHP configuration

- ODBC
  - Copy libodbsequoia.so in /usr/lib
  - Configure odbc.ini (look at the examples)
- MySQL native library
  - Install libMySequoia rpm or dbm package
  - LD_PRELOAD=/usr/lib/libmysequoia.so your_program
- Perl
  - Use DBD:JDBC module

Configuring Sequoia with Derby Network server

- Virtual database configuration file

```
<VirtualDatabase name="myDB">
  <Distribution>
      <MessageTimeouts/>
  </Distribution>
  …
  <DatabaseBackend name="derby1"
   driver="org.apache.derby.jdbc.ClientDriver"
   url=
  "jdbc:derby:net://localhost:1527/xpetstore;create=tru
  e;retrieveMessagesFromServerOnGetMessage=true;"
   connectionTestStatement="values 1">
      <ConnectionManager …/>
  </DatabaseBackend>
```

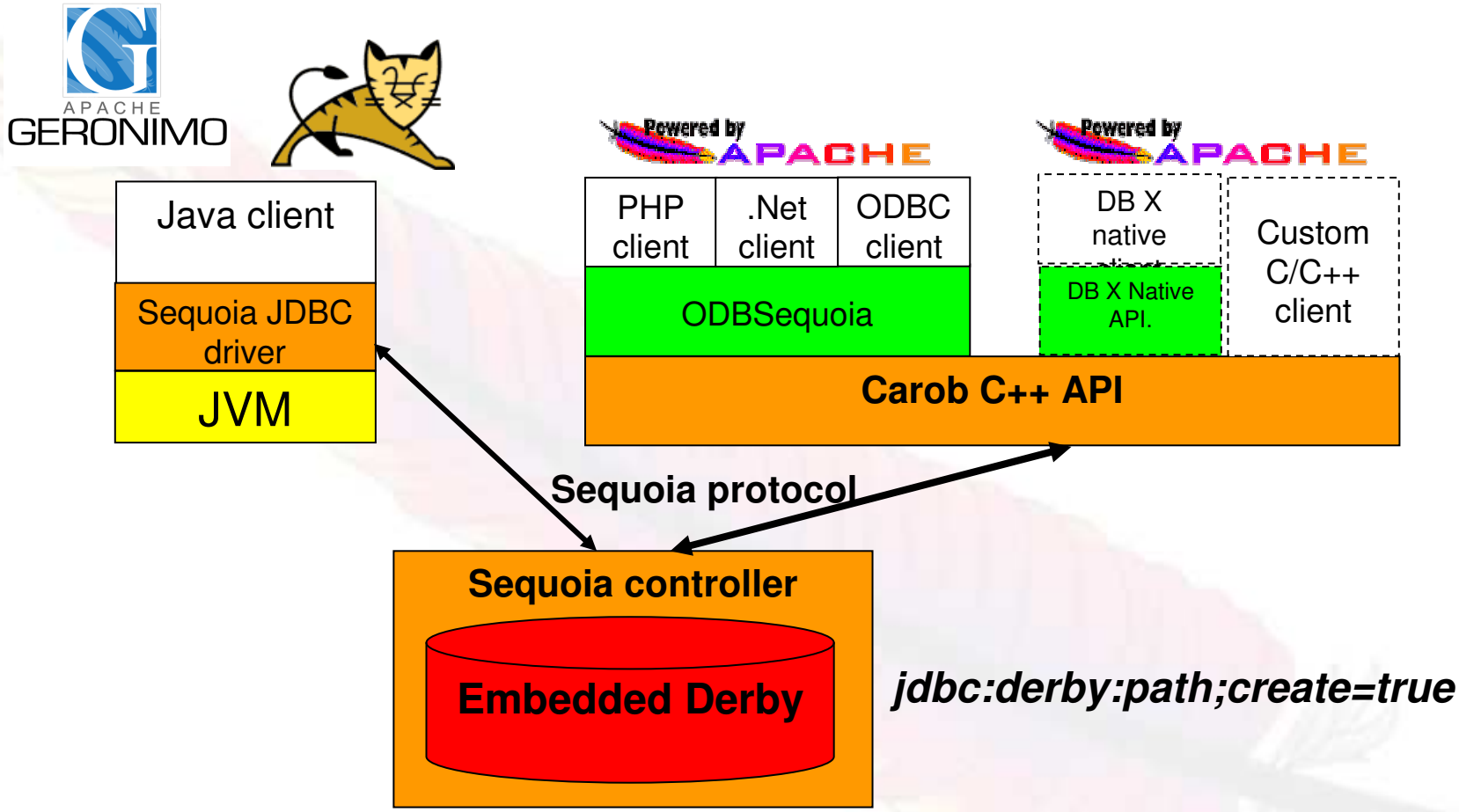
Configuring Sequoia Clustering with Derby (1/2)

```
<RequestManager>
  <RequestScheduler>
    <RAIDb-1Scheduler level="passThrough"/>
  </RequestScheduler>

  <RequestCache>
    <MetadataCache/>
    <ParsingCache/>
    <ResultCache granularity="table" />
  </RequestCache>

  <LoadBalancer>
    <RAIDb-1>
      <RAIDb-1-LeastPendingRequestFirst/>
    </RAIDb-1>
  </LoadBalancer>
```

Configuring Sequoia Clustering with Derby (2/2)

```
<RecoveryLog>
 <JDBCRecoveryLog
      driver="org.apache.derby.jdbc.ClientDriver “
      url="jdbc:derby:net://localhost:1529/xpetstore;
      create=true;retrieveMessagesFromServerOnGetMessage=true;"
      login="APP" password="APP">
   <RecoveryLogTable tableName="RECOVERY"
      idColumnType="BIGINT NOT NULL" sqlColumnName="sqlStmt“
      sqlColumnType="VARCHAR(8192) NOT NULL“
      extraStatementDefinition=",PRIMARY KEY (id)"/>
   <CheckpointTable tableName="CHECKPOINT"/>
   <BackendTable tableName="BACKENDTABLE"/>
   <DumpTable tableName=“DUMPTABLE”/>
  </JDBCRecoveryLog>
 </RecoveryLog>
</RequestManager>
</VirtualDatabase>
```

Sequoia to access Derby embedded



Outline

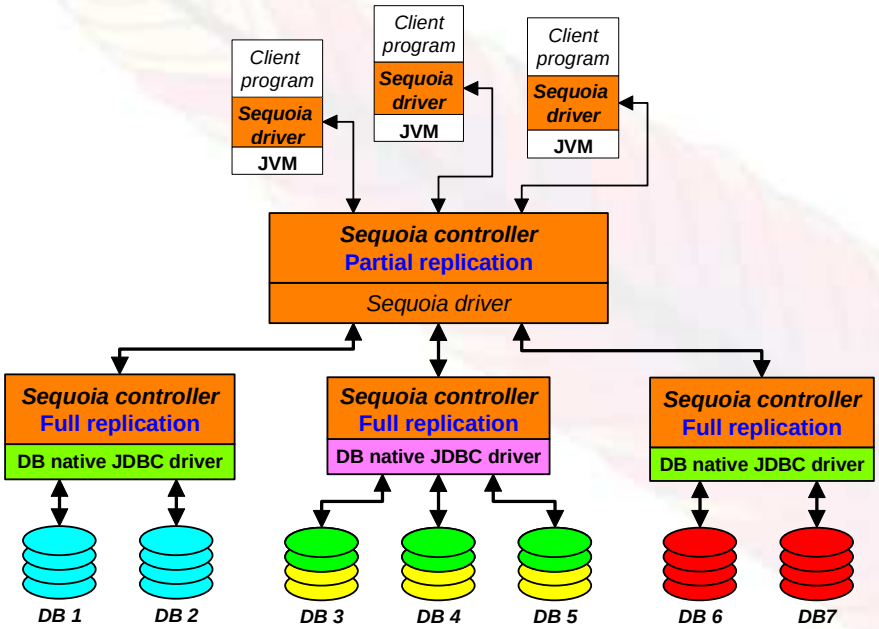
- RAIDb
- Sequoia
- Building an HA solution
- Scalability
- High availability

Sequoia vertical scalability

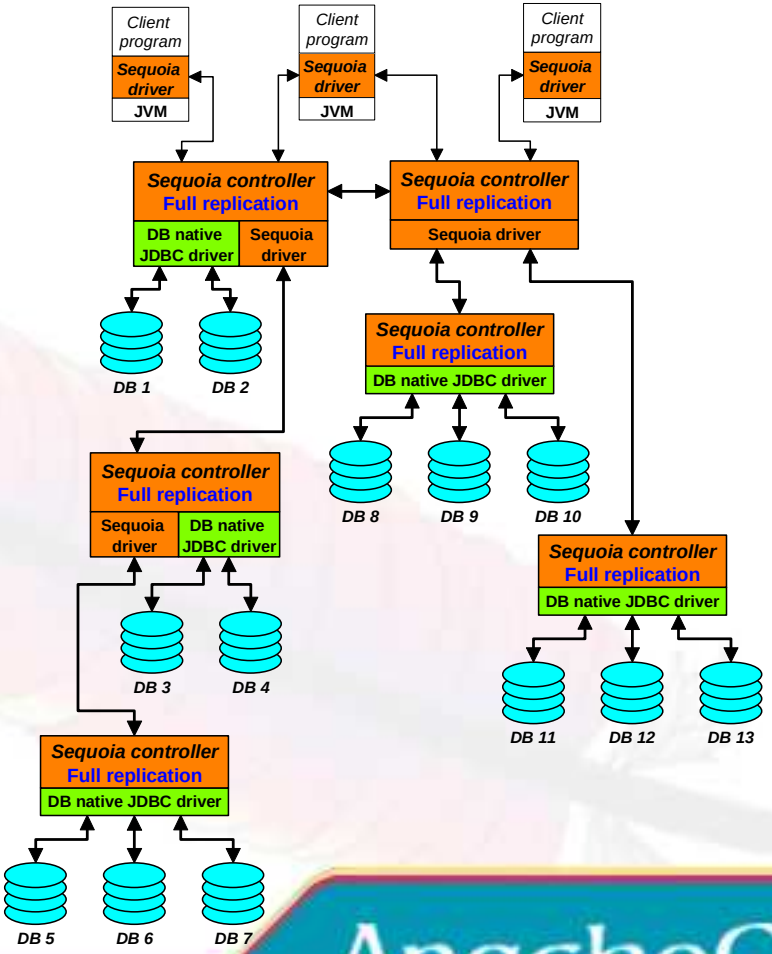
- allows nested RAIDb levels
- allows tree architecture for scalable write broadcast
- Sequoia driver re-injected in Sequoia controller

Advanced Configurations

Vertical Scalability

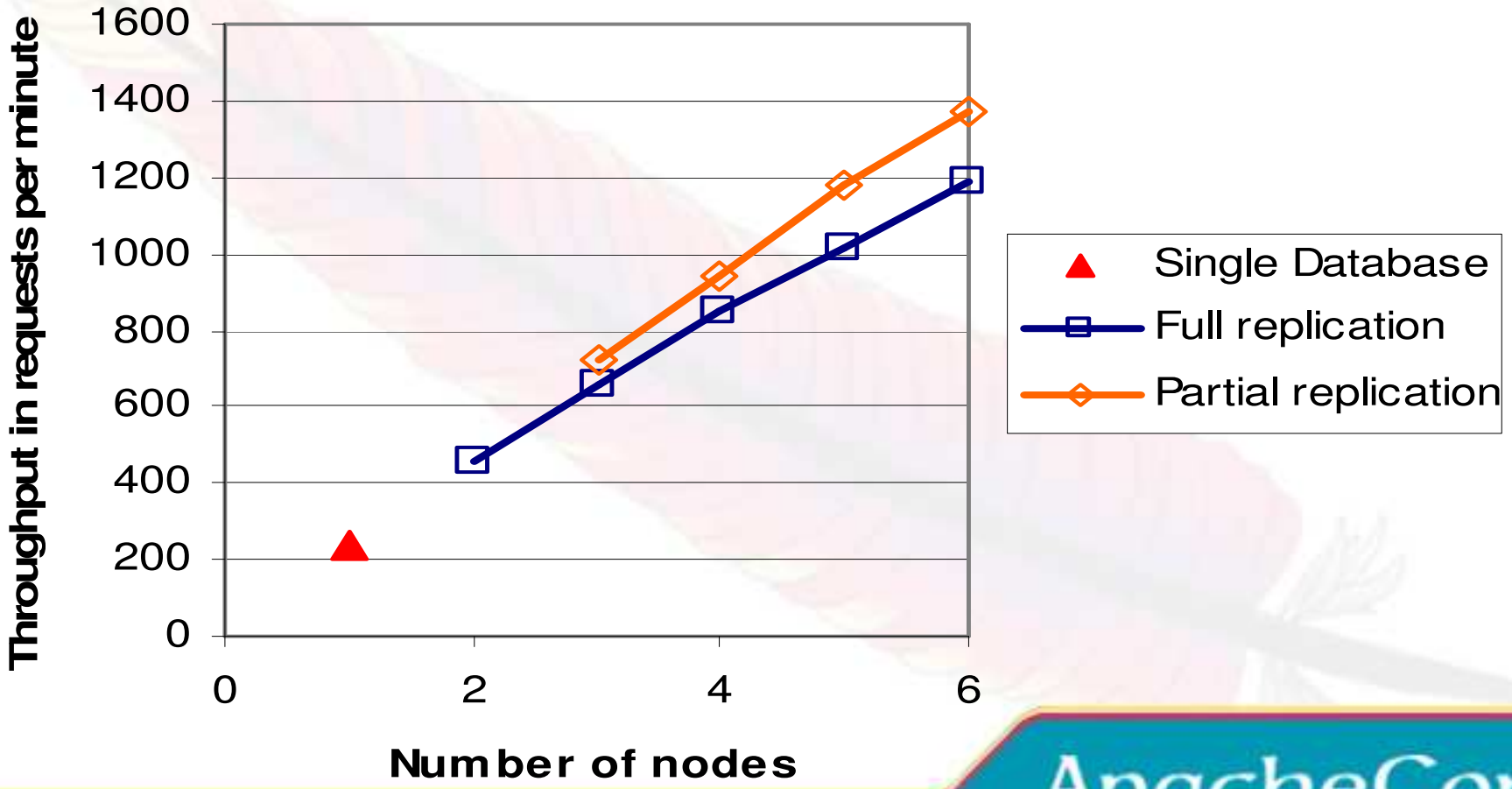


Mixed Horizontal and Vertical Scalability



TPC-W benchmark (Amazon.com)

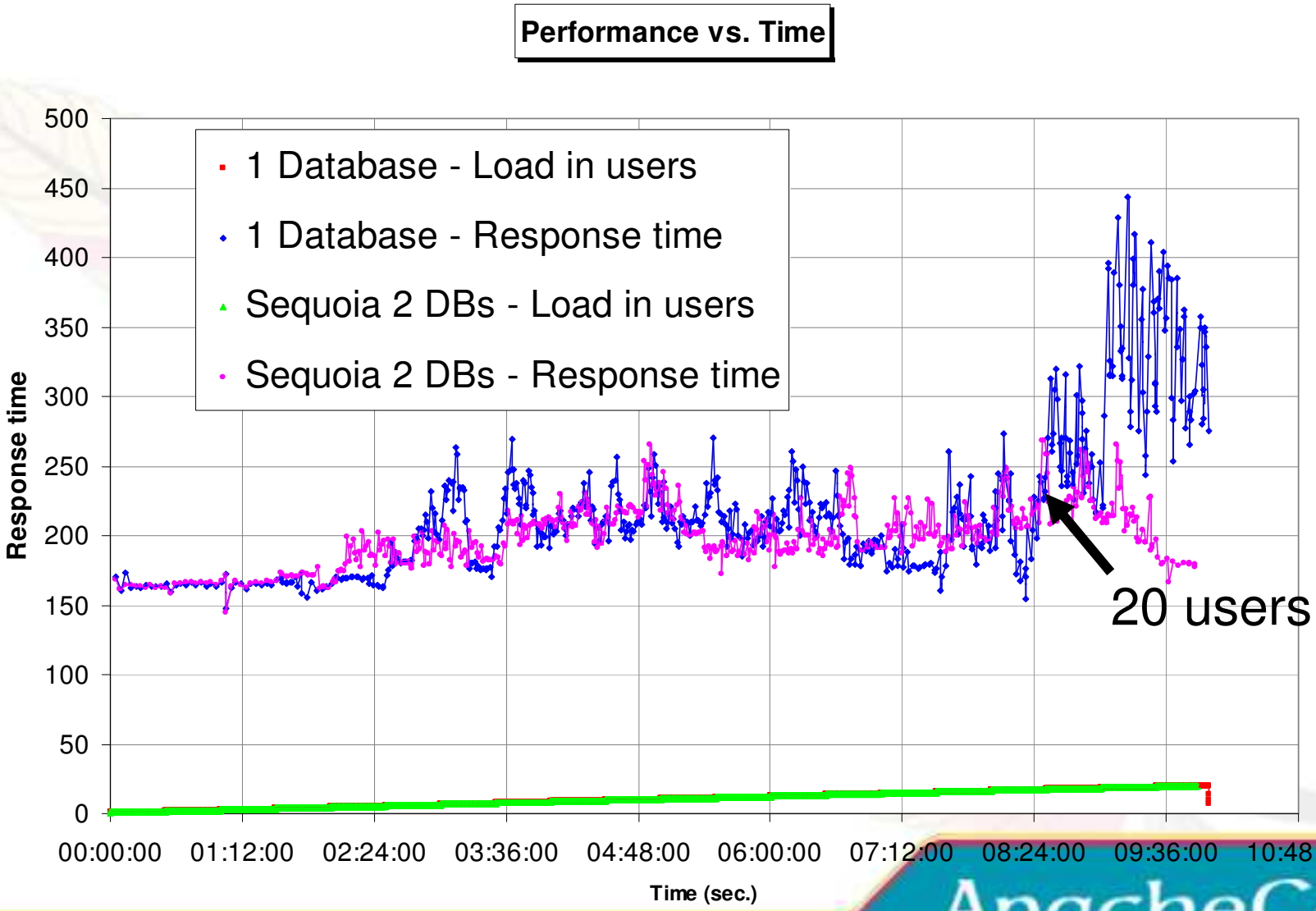
- Nearly linear speedups with the shopping mix



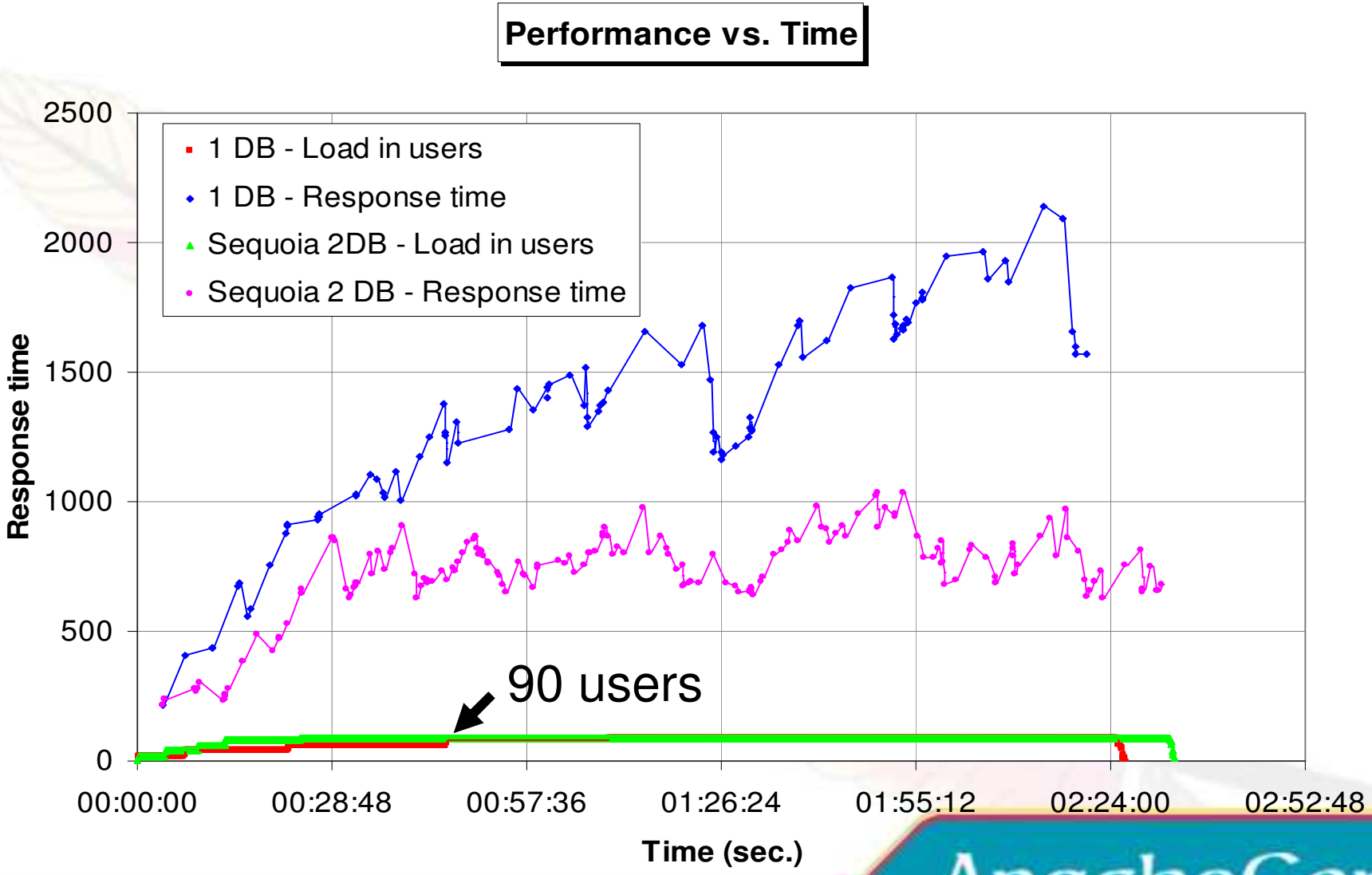
Performance vs Scalability

- Sequoia is not a parallel database
  - No parallelization of query execution plan
  - Query do not go faster when database is not loaded
- Sequoia distributes the load evenly
  - Constant response time when load increases
  - Better throughput when load surpasses capacity of a single database

Understanding scalability (1/2)



Understanding scalability (2/2)

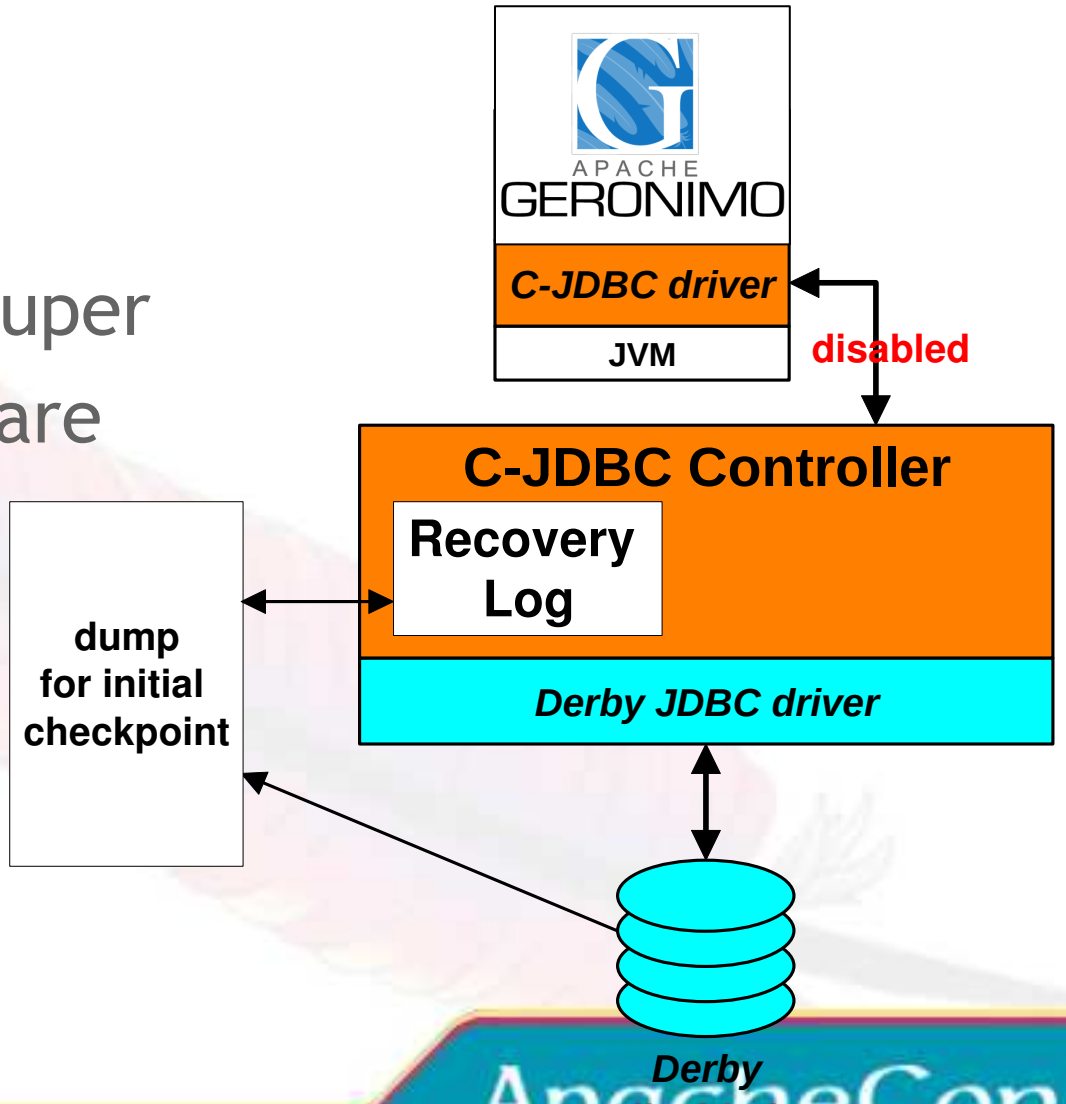


Outline

- RAIDb
- Sequoia
- Building an HA solution
- Scalability
- High availability

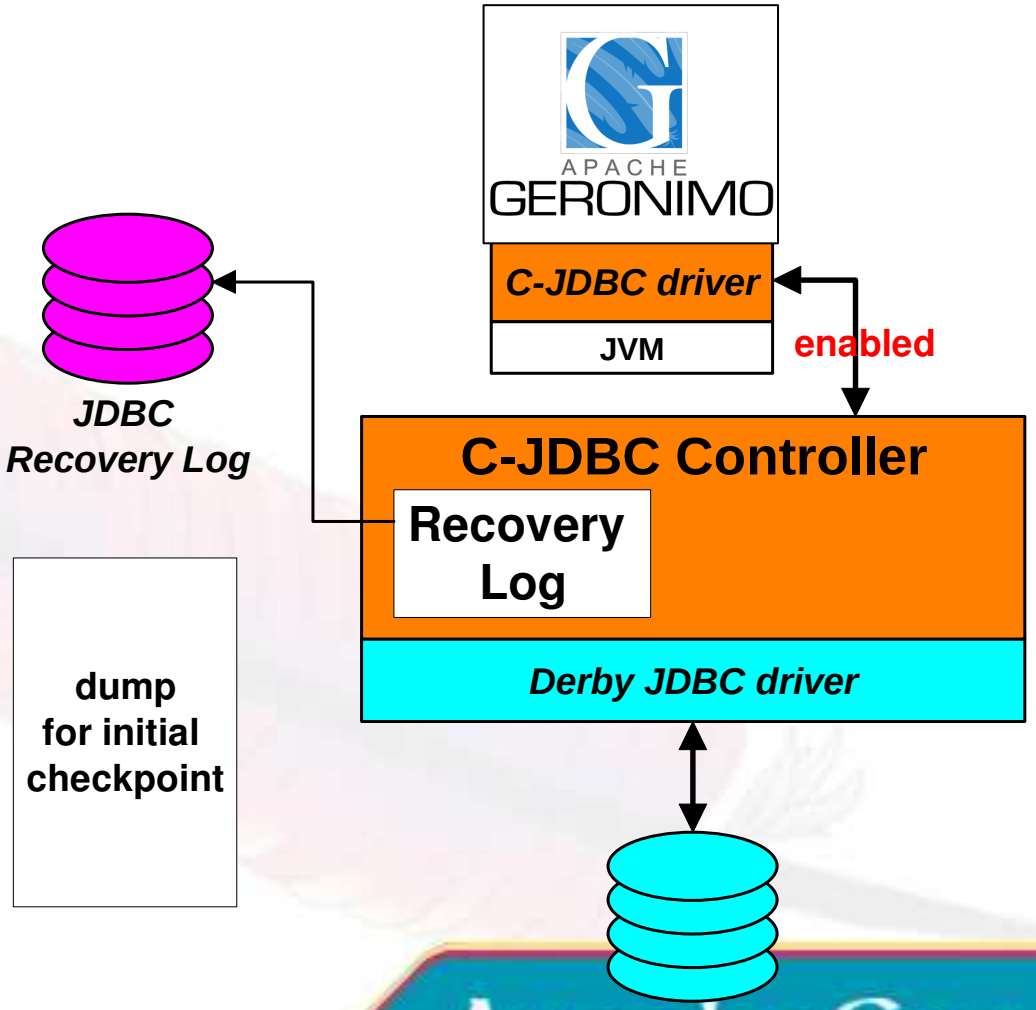
Initializing the cluster

- Dump initial Derby database using backuper
- RecoveryLog tables are initialized



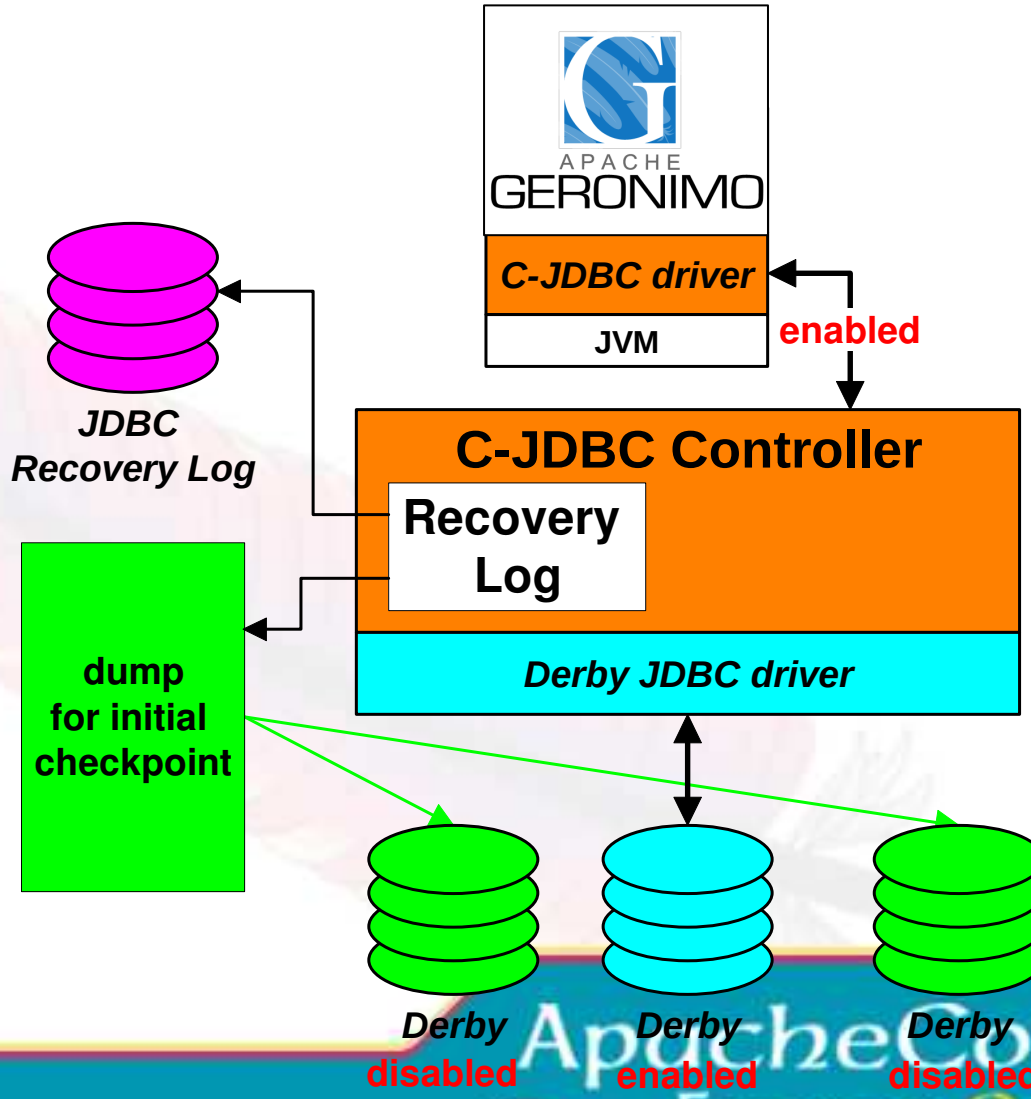
Logging

- Backend is enabled
- All database updates are logged (SQL statement, user, transaction, …)



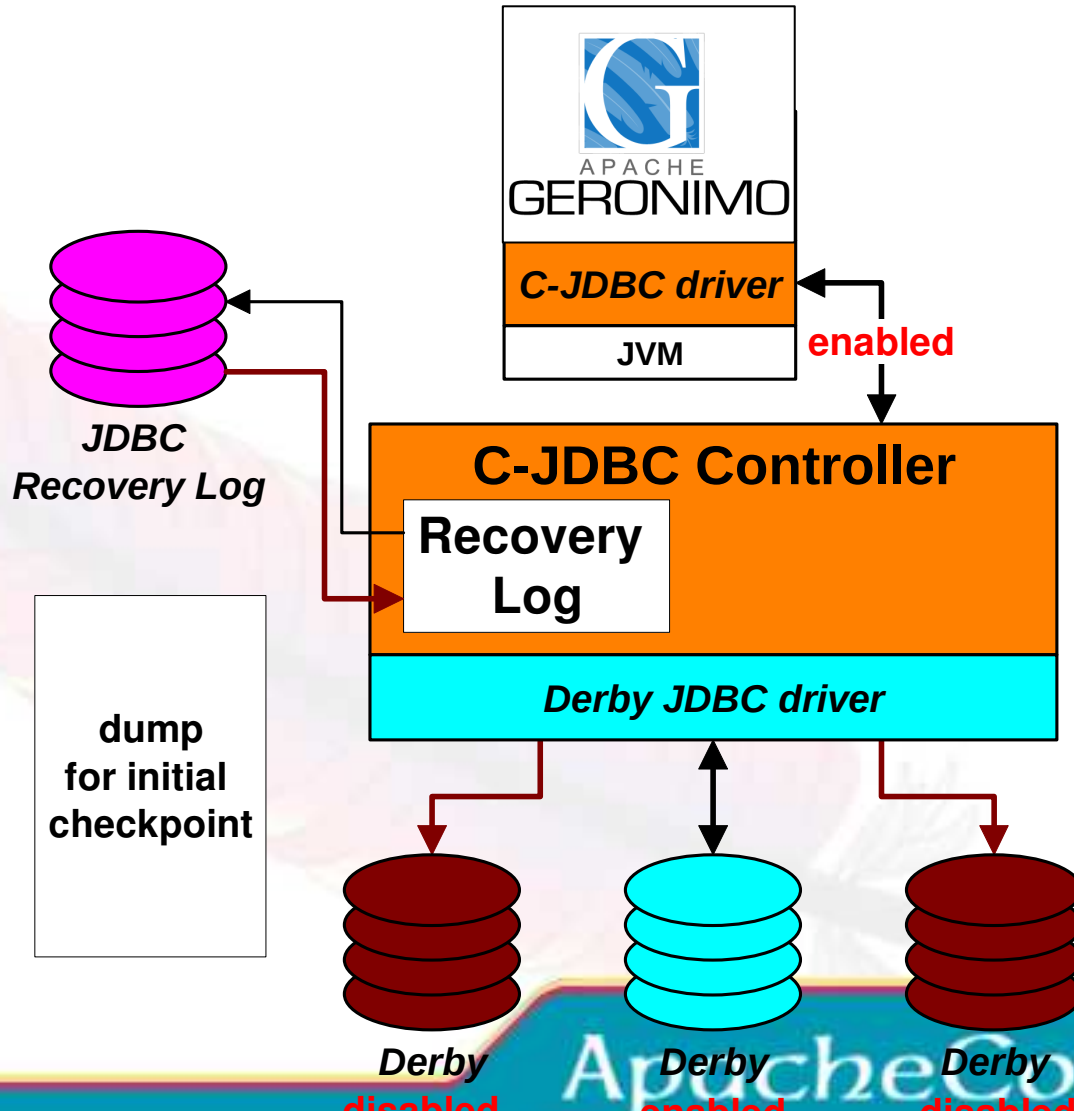
Adding new backends 1/3

- Add new backends while system online
- Restore dump



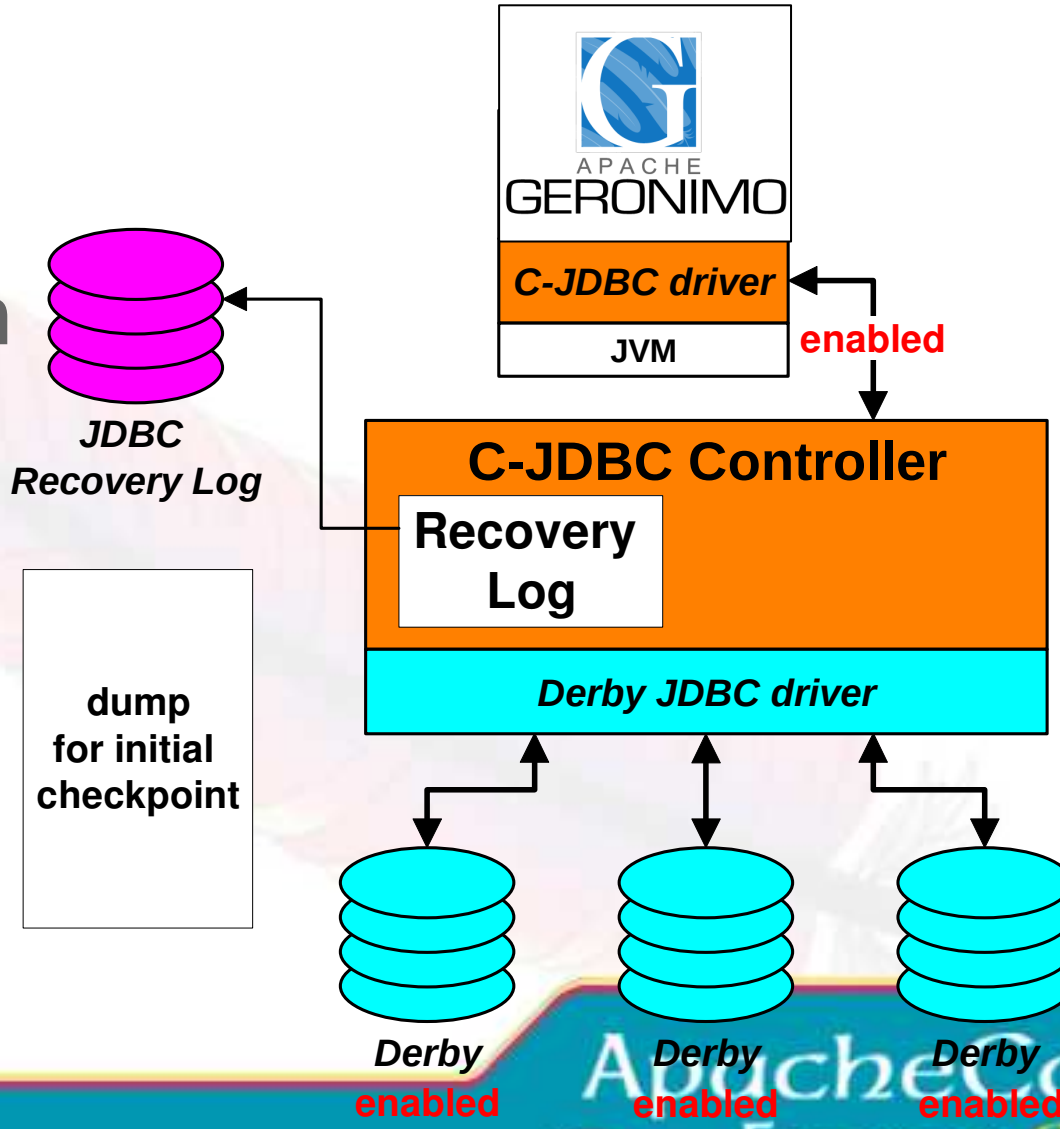
Adding new backends 2/3

- Replay updates from the log



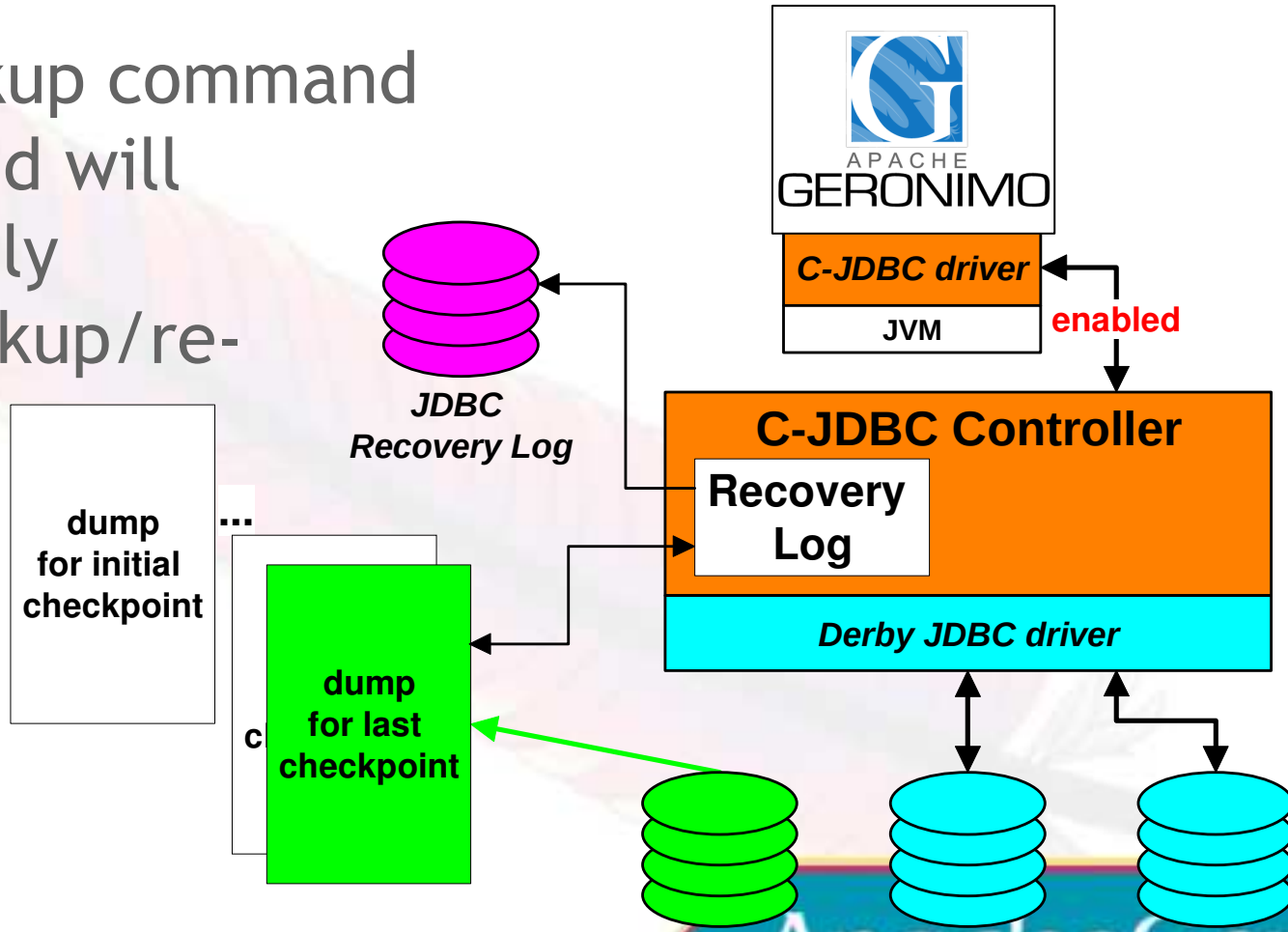
Adding new backends 3/3

- Auto-enable backends when done



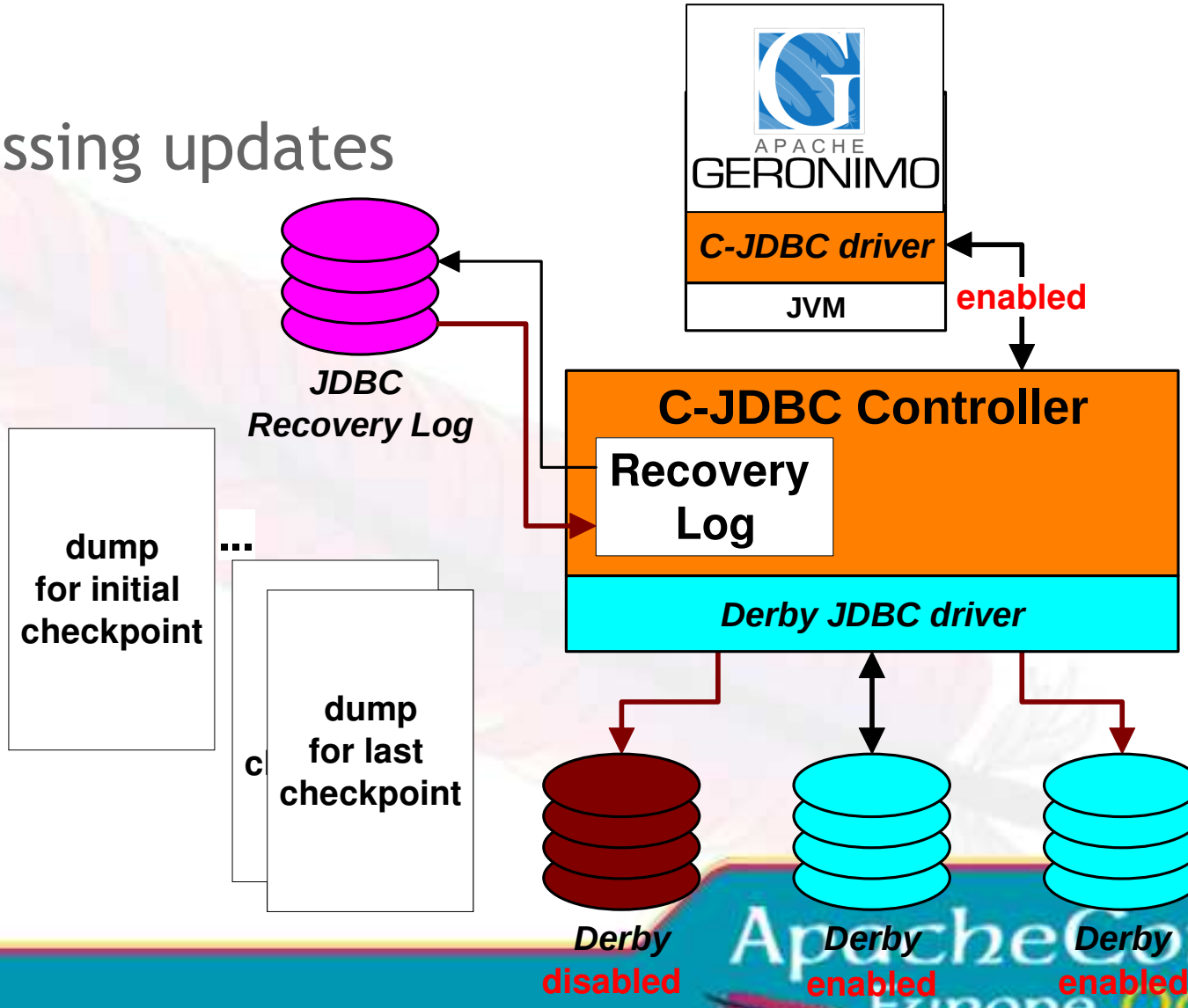
Taking new dumps (1/3)

- Calling backup command on a backend will automatically disable/backup/re-enable



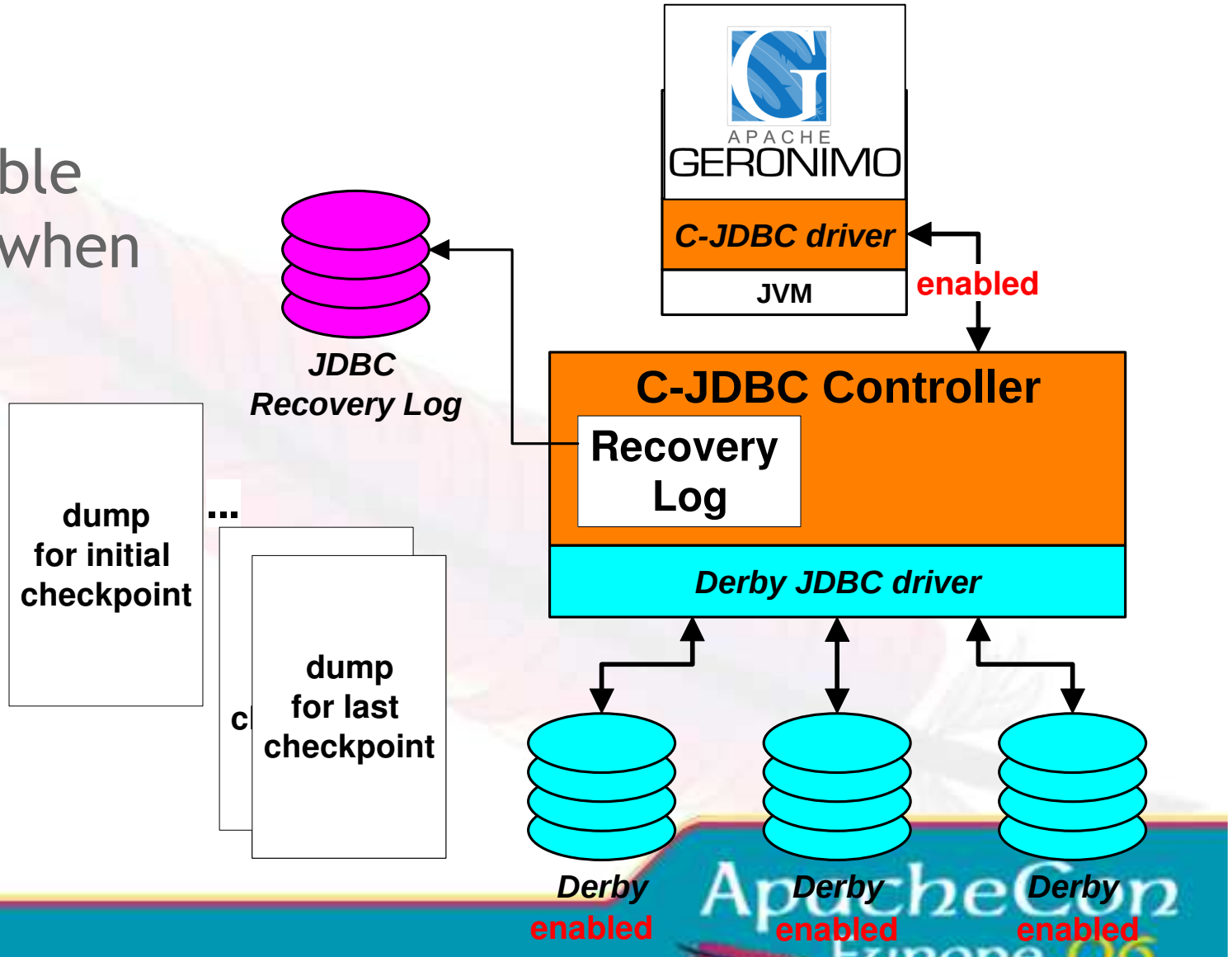
Taking new checkpoints (2/3)

- Replay missing updates from log



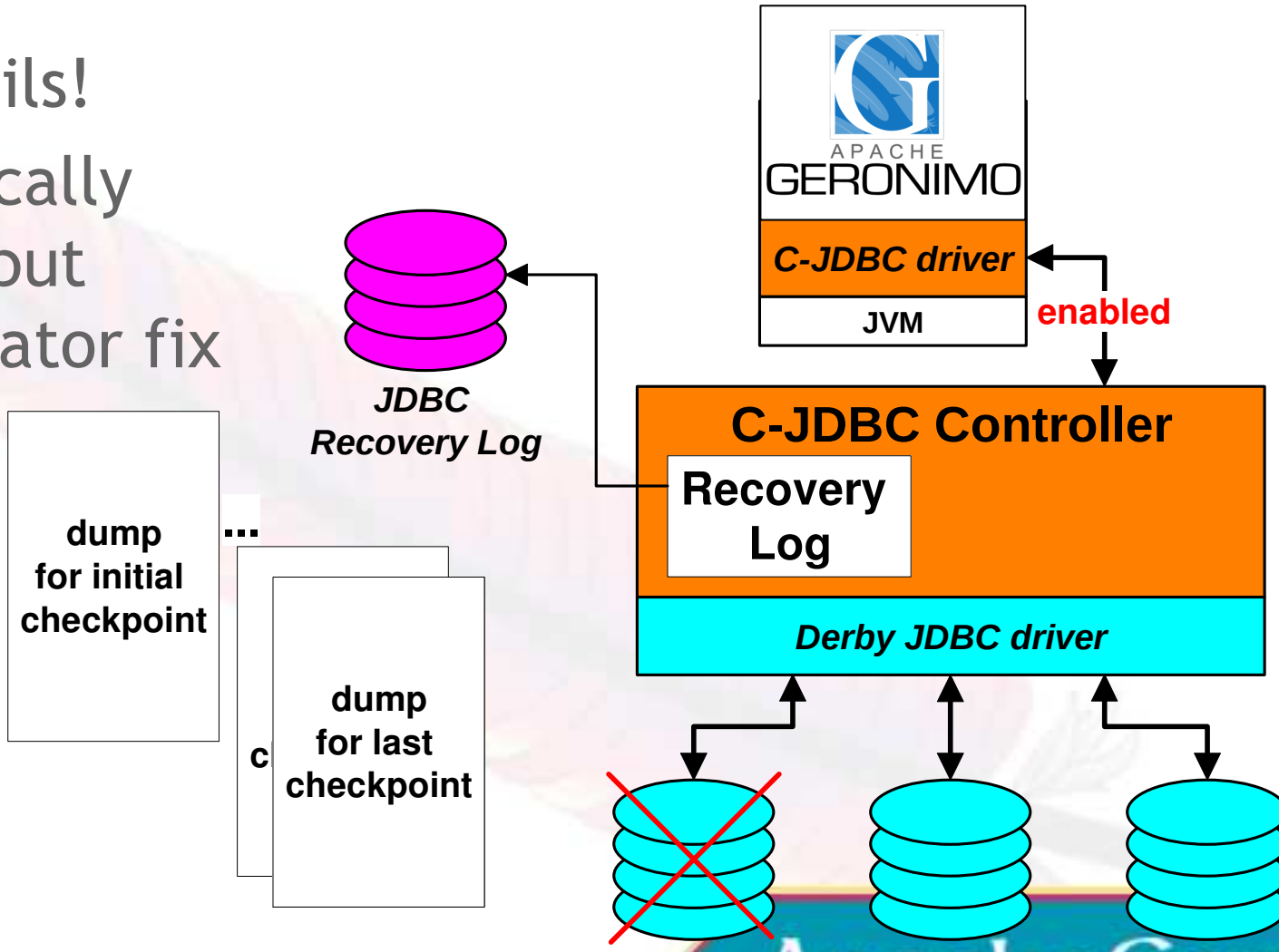
Making new checkpoints (3/3)

- Auto-enable backend when done



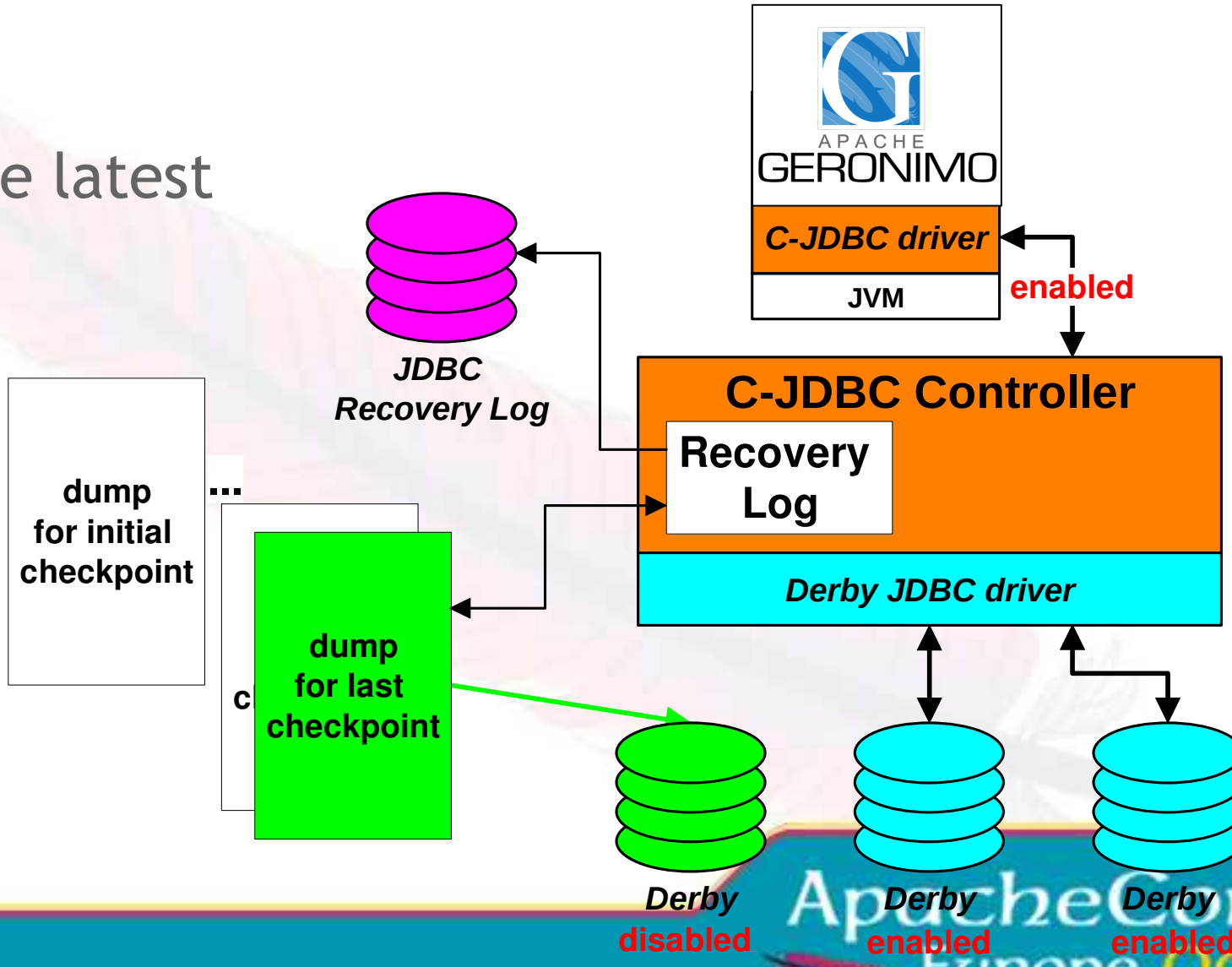
Handling failures

- A node fails!
- Automatically disabled but administrator fix needed



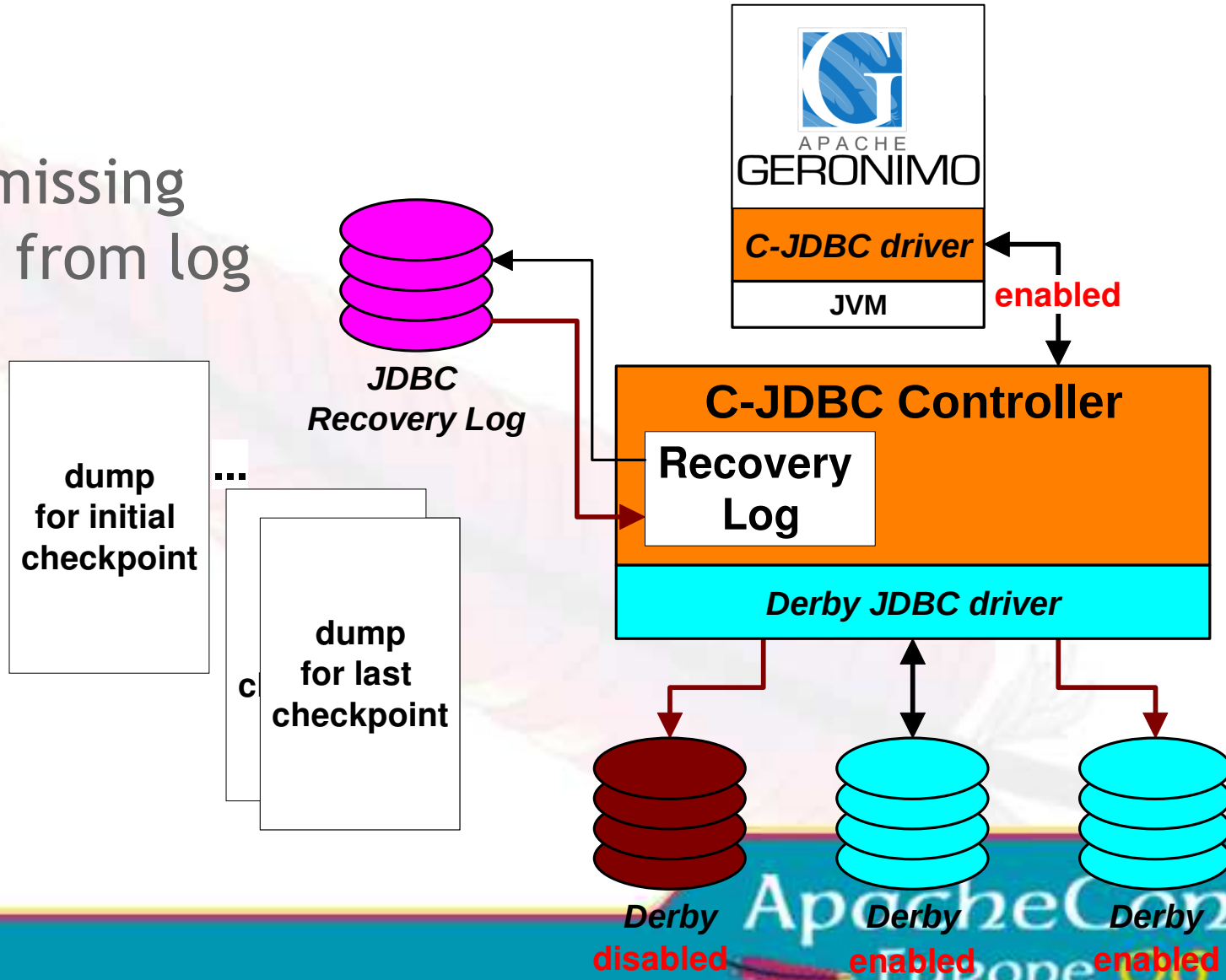
Recovery 1/3

- Restore latest dump



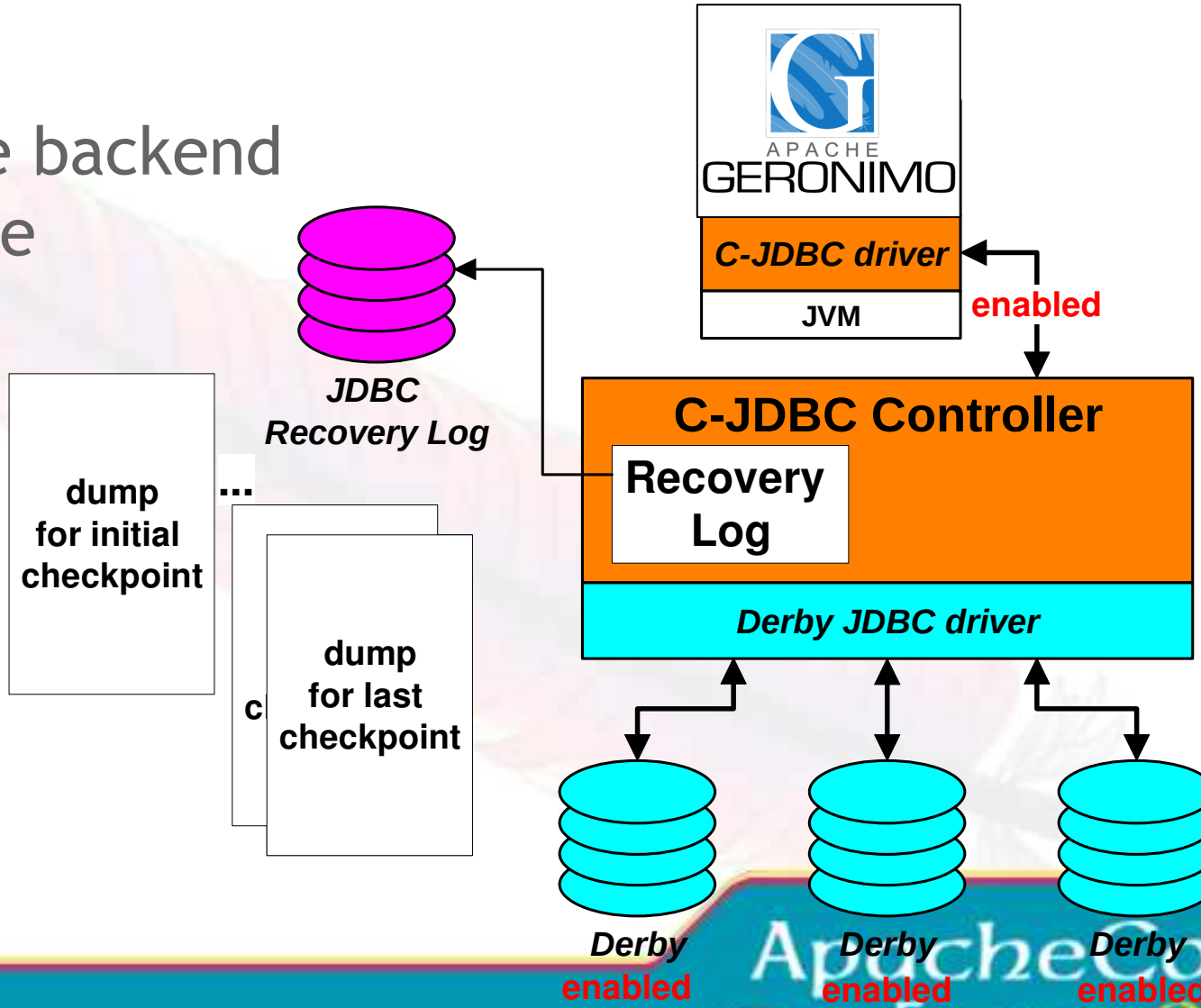
Recovery 2/3

- Replay missing updates from log



Recovery 3/3

- Re-enable backend when done



Current limitations

- Distributed joins
- Some JDBC 3.0 extensions
- XA support through XAPool only
- Triggers and updateable views supported with limitations (fully available in Sequoia 3.0)
- WAN network partition and reconciliation not supported

Summary

- Multi-tier HA
  - replication needed at each tier
  - Appia group communication
- Sequoia
  - high availability with fully transparent failover
  - performance scalability
- Visit http://www.continuent.org

Q&A

Thanks to all users and contributors ...

http://www.continuent.org