Scaling Hibernate Applications with Postgres

Bruce Momjian
Jim Mlodgenski
JbossWord 2009
www.enterprisedb.com
Objectives of this Session

• To show some the typical scaling techniques used by Postgres
• To demonstrate how Hibernate can leverage those techniques
• To highlight some issues using Hibernate with Postgres and how to overcome them
Who Are We?

Bruce Momjian
Sr Database Architect
EnterpriseDB Corporation

Jim Mlodgenski
Chief Architect
EnterpriseDB Corporation

About EnterpriseDB

• Award-winning open source database company
• World’s largest concentration of PostgreSQL expertise
• Headquartered in Westford, MA, USA
• 300+ customers including Sony, FTD, hi5 Networks, NetApp, FAA, LexisNexis, St. Jude Children’s Hospital
• 35+ partners including Red Hat, IBM, Compiere
About Hibernate

- Object-relational persistence framework
- Supports collections, object relations, composite types
- HQL query language, caching, JMX support

Diagram:

Application Tier

Hibernate

Connectivity

Data Tier

Persistent Objects

Transactions
Connections
Sessions
Queries
Criteria

JDBC
JNDI
JTA
About PostgreSQL and Postgres Plus

- Powerful open source object-relational database
- 20+ years of global community development
- Enterprise-class functionality

<table>
<thead>
<tr>
<th>Feature</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully ACID compliant</td>
<td>Views</td>
</tr>
<tr>
<td>Foreign Keys</td>
<td>Triggers</td>
</tr>
<tr>
<td>Joins</td>
<td>Stored Procedures (multiple languages)</td>
</tr>
<tr>
<td>Multi-version Concurrency Control</td>
<td>Point-in-Time Recovery</td>
</tr>
<tr>
<td>Tablespace</td>
<td>Asynchronous Replication</td>
</tr>
<tr>
<td>Nested transactions</td>
<td>Heap Only Tuples</td>
</tr>
<tr>
<td>Online/hot backups</td>
<td>Write-ahead logging (fault tolerance)</td>
</tr>
</tbody>
</table>

- Mature Hibernate dialect
Common Scaling Problems

- Application Server overloaded
- Database overloaded
- Slow database queries

Developers: “this database is a dog”

DBAs: “quit sending all that inefficient SQL”
Three popular scaling strategies:

1. Add more hardware to scale horizontally or vertically
2. Improve the performance of individual operations
   - The faster an operation completes, the sooner the system is available for other work
3. Eliminate non-critical operations to reserve resources
   - Reduce the overhead of business logic execution to keep resources available for handling user loads
What's Hibernate?!?! 

• Many database administrators have never heard of it  
  – “Isn't that what bears do?”
• When they do hear of it, they rarely like it
• The business and development benefits are not always readily apparent to database administrators
Fetching Strategies

- When loading objects with a parent child relationship, there are a large number of selects performed on the database
  - The N+1 problem

```
LOG: execute S_1: BEGIN
LOG: execute <unnamed>: select fdgroup0_.fdgrp_cd as fd...
DETAIL: parameters: $1 = '0100'
LOG: execute <unnamed>: select fooddeses0_.fdgrp_cd as ...
DETAIL: parameters: $1 = '0100'
LOG: execute S_2: COMMIT
LOG: execute S_1: BEGIN
LOG: execute <unnamed>: select fdgroup0_.fdgrp_cd as fd...
DETAIL: parameters: $1 = '0100'
LOG: execute <unnamed>: select fooddeses0_.fdgrp_cd as ...
DETAIL: parameters: $1 = '0100'
LOG: execute S_2: COMMIT
LOG: execute <unnamed>: select fooddeses0_.fdgrp_cd as ...
DETAIL: parameters: $1 = '0100'
LOG: execute S_2: COMMIT
```
• In many cases, the default fetching settings of “lazy” and “select” work well
  – When working with related tables that have many child records that are needed, this is inefficient

```java
@Entity
@Table(name = "fd_group")
public class FdGroup implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "fdgrp_cd")
    private String fdgrpCd;
    ...
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "fdgrpCd")
    @org.hibernate.annotations.Fetch(org.hibernate.annotations.FetchMode.JOIN)
    private Collection<FoodDes> foodDesCollection;
```
Caching

- The number of reads hitting the database are exceeding what a single server can handle.
Caching (cont.)

• A second level cache eliminates a lot of interaction with the database

• There are many options depending on the needs on the application
  – EhCache
  – OSCache
  – Terracotta
  – Infinispan

• Be aware of direct database changes causing cache invalidations
Connection Pooling

- As the number of concurrent users increase, the database has trouble dealing with all of the new connections
  - Postgres is process based so a new database connection is very expensive
  - This expense is magnified in a Windows environment
Connection Pooling (cont.)

• PgBouncer provides an easy to administer connection pool for Postgres
  – This will be transparent to all Hibernate configurations

• Unfortunately, all data must pass through an additional layer adding overhead

• Ideal for use when clustering the application servers and a common pool across all servers is needed
Connection Pooling (cont.)

- A connection pool within the application server eliminates the overhead of a layer
  - C3P0, DBCP, and Proxool can all serve this purpose
  - Easily setup using the standard configuration files

```
hibernate.connection.driver_class = org.postgresql...
hibernate.connection.url = jdbc:postgresql://lo...
hibernate.connection.username = postgres
hibernate.connection.password = password
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = org.hibernate.dialect.Postg...
```
Updates are Causing Havoc

• The update statements send all of the columns even if they did not change
  – The Postgres logs show the following lines when just updating the column “filler” to “HelloWorld”

```
LOG:  execute <unnamed>: update tellers set bid=$1, filler=$2, tbalance=$3 where tid=$4
DETAIL:  parameters: $1 = '1', $2 = 'HelloWorld', $3 = '0', $4 = '1'
LOG:  execute S_2: COMMIT
```

• This causes several problems...
• Why is this a problem?
  – The application server and the database are doing more work than necessary
    • A much longer SQL statement must be constructed by Hibernate
    • There is more network traffic as this is sent to the database
    • The database needs to parse a much bigger SQL statement

• Effect of curing this symptom (Minor)
  – Modern hardware has the CPU and network bandwidth to handle this additional overhead
  – Probably only noticeable with a large number of transactions
Updates are Causing Havoc (cont.)

• Why is this still a problem?
  – Foreign Key Integrity checks are fired unnecessarily
    • Adds significantly more load to the database for each update
    • This grows for each Foreign Key on the table
    • The amount of work increases as the size of the related table increases

• Effect of curing this symptom (Moderate)
  – If there are only a few Foreign Keys, this may not affect performance
  – If the related tables are small and updates are somewhat frequent, the buffers will be in the database cache
Updates are Causing Havoc (cont.)

• Really, why is this a problem?
  – None of the updates can leverage HOT if there are indexes on the table
    • This can cause table and index bloat for frequently updated tables
    • Adds additional maintenance overhead to the database to clean up the bloat

• Effect of curing this symptom (Major)
  – Performance steadily degrades on frequently updated tables
Updates are Causing Havoc (cont.)

• How is this solved?
  – Use the Dynamic Update annotation

@Entity
@Table(name = "tellers")
@Entity(
    selectBeforeUpdate = true,
    dynamicInsert = true, dynamicUpdate = true)
@NamedQueries(
    {@NamedQuery(name = "Tellers.findAll",
        query = "SELECT t FROM Tellers t"),
     @NamedQuery(name = "Tellers.findByTid",
        query = "SELECT t FROM Tellers t WHERE t.tid = :tid"),
     @NamedQuery(name = "Tellers.findByBid",
        query = "SELECT t FROM Tellers t WHERE t.bid = :bid"),
     @NamedQuery(name = "Tellers.findByTbalance",
        query = "SELECT t FROM Tellers t WHERE t.tbalance = :tbalance")})
public class Tellers implements Serializable {

Table Partitioning

- A table is getting very large and the database administrator just partitioned the table, but now the application can no longer insert into the table.

```java
Could not synchronize database state with session
org.hibernate.StaleStateException: Batch update returned unexpected row count from update [0]; actual row count: 0; expected: 1
    at org.hibernate.jdbc.Expectations$BasicExpectation.checkBatched(Expectations.java:187)
    at org.hibernate.jdbc.Expectations$BasicExpectation.verifyOutcome(Expectations.java:150)
    at org.hibernate.jdbc.BatchingBatcher.checkRowCounts(BatchingBatcher.java:68)
    at org.hibernate.jdbc.BatchingBatcher.doExecuteBatch(BatchingBatcher.java:48)
    at org.hibernate.jdbc.AbstractBatcher.executeBatch(AbstractBatcher.java:246)
    at org.hibernate.engine.ActionQueue.executeActions(ActionQueue.java:237)
    at org.hibernate.engine.ActionQueue.executeActions(ActionQueue.java:141)
```
Table Partitioning (cont.)

• What is the problem?
  – From the database command line, psql, a row can be inserted
  – Hibernate is expecting the row count to be 1, but Postgres is returning 0

• How can the row be inserted into the database but return a message that 0 rows have be updated?
  – Postgres uses inherited tables and triggers to implement table partitioning
  – No rows are actually inserted into the base table, so the database is behaving correctly
Table Partitioning (cont.)

• How is this solved?
  – Change the Postgres trigger function to return a row

```sql
CREATE OR REPLACE FUNCTION accounts_insert_trigger()
    RETURNS trigger AS
$BODY$
DECLARE
    ret accounts%ROWTYPE;
BEGIN
    ret.aid = -1;
    .
    .
    RETURN ret;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE
```

• But there are some side affects to this approach
Table Partitioning (cont.)

- Returning a row from the trigger function leaves a row in the base table which is not wanted
  - Create an After trigger to remove the row

```sql
CREATE OR REPLACE FUNCTION accounts_insert_clean_trigger()
RETURNS trigger AS
$BODY$
BEGIN
    DELETE FROM accounts WHERE aid = -1;
    RETURN NULL;
END;
$BODY$
LANGUAGE 'plpgsql' VOLATILE
```

- This is not very scalable. 2 inserts and 1 delete to create a single row is not practical
Table Partitioning (cont.)

• What is a better way to solve this?
  – Use the SQLInsert annotation to suppress the row count check when inserting the row

```java
@Entity
@Table(name = "accounts")
@SQLInsert(
    sql="INSERT INTO accounts (abalance, bid, filler, aid)VALUES (?, ?, ?, ?)",
    check=ResultCheckStyle.NONE)
@NamedQueries({
    @NamedQuery(name = "Accounts.findAll",
        query = "SELECT a FROM Accounts a"),
    @NamedQuery(name = "Accounts.findByAid",
        query = "SELECT a FROM Accounts a WHERE a.aid = :aid"),
    @NamedQuery(name = "Accounts.findByBid",
        query = "SELECT a FROM Accounts a WHERE a.bid = :bid")})
public class Accounts implements Serializable {
```
Data Model

- There are a large number of joins necessary to get basic information for the application.
- The data model is extremely flexible so any “widget” can be stored.
Data Model (cont.)

• Denormalization is not always bad
  – But be careful not to have multiple versions of the truth
• Reduces the number of database calls or joins to get the data for the user
• A common technique is using Materialized Views or OLAP cubes
  – These are not native constructs in Postgres
  – But...they can be created with native Postgres constructs like Rule and Triggers
Database Advanced Features

• Some of the advanced features in Postgres will speed up the queries
  – Windowing Functions, Hierarchical Queries, Spatial, etc
• HQL does not support it and throws an exception

unexpected token: OVER near line 1, column 121 [SELECT insurance.insur...
• Send native SQL to the database

```java
String SQL_QUERY = "SELECT insurance_name, id, invested_amount, avg(i...
   + "invested_amount - avg(invested_amount) OVER(PARTI...
   + "FROM insurance ");
Query query = session.createQuery(SQL_QUERY)
  .addScalar("insurance_name", Hibernate.STRING)
  .addScalar("id", Hibernate.LONG)
  .addScalar("invested_amount", Hibernate.LONG)
  .addScalar("a", Hibernate.DOUBLE)
  .addScalar("diff", Hibernate.DOUBLE);
```

• This adds database specific code inside the application
Database Advanced Features (cont.)

- Use a database view

```
CREATE VIEW insurance_diff AS
    SELECT insurance_name, id, invested_amount,
       avg(invested_amount) OVER(PARTITION BY insurance_name),
       invested_amount - avg(invested_amount)
       OVER(PARTITION BY insurance_name) AS diff
    FROM insurance
```

- Allows the use of standard HQL
  - Prevents database specific code inside the application
- Puts the database specific code inside the database
Replication

- The number of reads hitting the database are still exceeding what a single server can handle.
- Sometimes horizontal scaling is necessary
Replication (cont.)

- Replicate using Slony to additional servers
  - Asynchronous single master multiple slave
- Load Balance reads using pgpool-II
- All writes go to the master node
- Connection pooling is also handled by pgpool-II
• The possibility of row version problems exists
  – The application can read a row from the slave server before a new version is replicated
• Optimistic Concurrency Control can be used to solve these problems
  – Row versions are handled by the application by adding a version column to the tables

```java
public class Accounts implements...
private static final long ...  
@Id  
@Basic(optional = false)
@Column(name = "aid")
private Integer aid;
@Column(name = "bid")
private Integer bid;
@Column(name = "abalance")
private Integer abalance;
@Column(name = "filler")
private String filler;
@Version
@Column(name = "version")
private Integer version;
```
Summary

- Fetching strategies
- Caching
- Setup connection pooling
- Reduce the overhead of updates
- Partition large tables
- Optimize the data model
- Use advanced database features
- Replication
Thank you.

Questions?