Java Technology Day

MySQL for Developers

Carol McDonald, Java Architect

Outline

- Storage engines
- Schema
- Normalization
- Data types
- Indexes
- Know your SQL
- Using Explain
- Partitions
- JPA lazy loading
- Resources

Why is it significant for a developer to know MySQL?

Generics are inefficient

take advantage of MySQL's strengths

understanding the database helps you develop better-performing applications

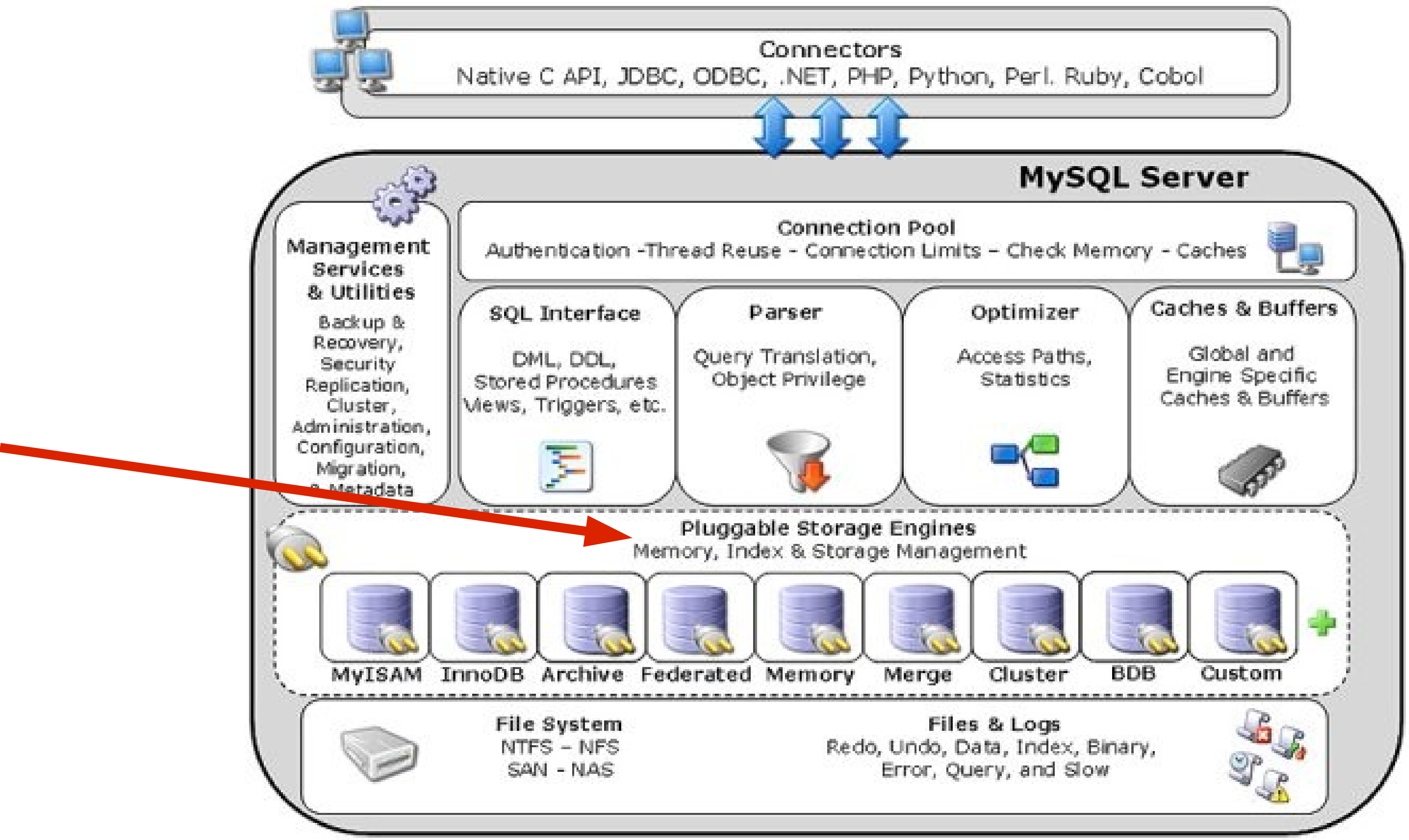
better to design a well-performing database-driven application from the start

...than try to fix a slow one after the fact!

MySQL Pluggable Storage Engine Architecture

MySQL supports several storage engines that act as handlers for different table types

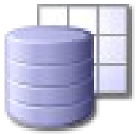
No other database vendor offers this capability



What makes engines different?

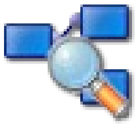
Concurrency: table lock vs row lock

right locking can improve performance.



Storage: how the data is stored on disk

size for tables, indexes



Indexes: improves search operations

Memory usage:

Different caching strategies

Transactions support

not every application needs transactions

So...

As a developer, what do I need to know about storage engines, without being a MySQL expert?

keep in mind the following questions:

What **type of data** will you be storing?

Is the data constantly **changing**?

Is the data **mostly** logs (**INSERT**s)?

requirements for **reports**?

Requirements for **transactions**?

MyISAM Pluggable Storage engine

Default MySQL engine

high-speed Query and Insert capability

insert uses shared read lock

updates,deletes use table-level locking, slower

full-**text indexing**

Non-transactional

good choice for :

read-mostly applications

that don't require transactions

Web, data warehousing, logging, auditing

InnoDB Storage engine in MySQL

Transaction-safe and ACID compliant

good query performance, depending on indexes

row-level locking, MultiVersion Concurrency Control (MVCC)

allows fewer row locks by keeping data snapshots

no locking for SELECT (depending on isolation level)

high concurrency possible

uses more disk space and memory than ISAM

Good for Online transaction processing (OLTP)

Lots of users: Slashdot, Google, Yahoo!, Facebook, etc.

Memory Engine

Entirely in-memory engine

stores all data in **RAM** for **extremely fast** access

Hash index used by default

Good for

Summary and transient data

"lookup" or "mapping" tables,

calculated table counts,

for caching Session or temporary tables

Archive engine

Incredible insert speeds

Great compression rates

No UPDATEs

Ideal for **storing and retrieving** **large amounts** of **historical** data

audit data, **log** files,Web traffic records

Data that can never be updated

Storage Engines

Feature	MyISAM	Falcon	NDB	Archive	InnoDB	Memory
Storage limits	No	110TB	Yes	No	64TB	Yes
Transactions	No	Yes	Yes	No	Yes	No
Locking granularity	Table	MVCC	Row	Row	Row	Table
MVCC snapshot read	No	Yes	No	No	Yes	No
Geospatial support	Yes	Yes	No	Yes	Yes	No
Data caches	No	Yes	Yes	No	Yes	NA
Index caches	Yes	Yes	Yes	No	Yes	NA
Compressed data	Yes	No	No	Yes	No	No
Storage cost (relative to other engines)	Small	Med	Med	Small	Med	NA
Memory cost (relative to other engines)	Low	High	High	Low	High	High
Bulk insert speed	High	Med	High	Highest	Med	High
Replication support	Yes	Yes	Yes	Yes	Yes	Yes
Foreign Key support	No	Yes	No	No	Yes	No
Built-in Cluster/High-availability support	No	No	Yes	No	No	No

Dynamically add and remove storage engines.

Change the storage engine on a table with “ALTER TABLE …”

Does the storage engine *really* make a difference?

User Load	MyISAM Inserts Per Se	InnoDB Inserts Per Se	Archive Inserts Per Se
1	3,203.00	2,670.00	3,576.00
4	9,123.00	5,280.00	11,038.00
8	9,361.00	5,044.00	13,202.00
16	8,957.00	4,424.00	13,066.00
32	8,470.00	3,934.00	12,921.00
64	8,382.00	3,541.00	12,571.00

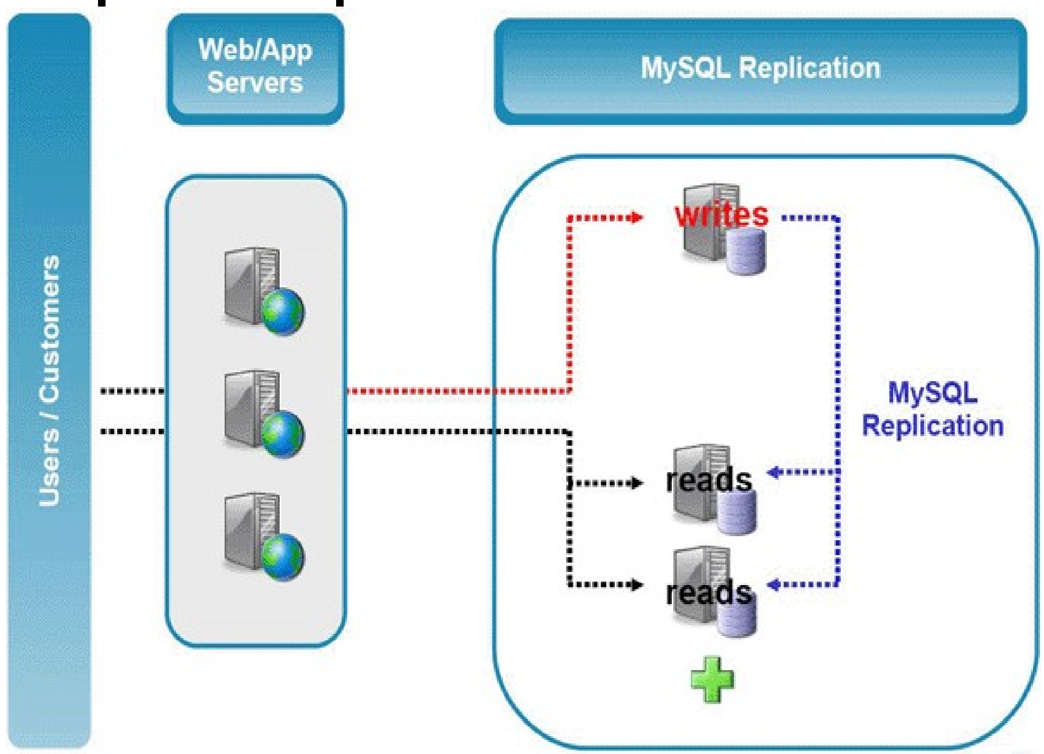
Using mysqlslap, against MySQL 5.1.23rc, the Archive engine has 50% more INSERT throughput compared to MyISAM, and 255% more than InnoDB

Pluggable storage engines offer Flexibility

You can use multiple storage engines in a single application

A storage engine for the same table on a slave can be different than that of the master

can greatly improve performance



master

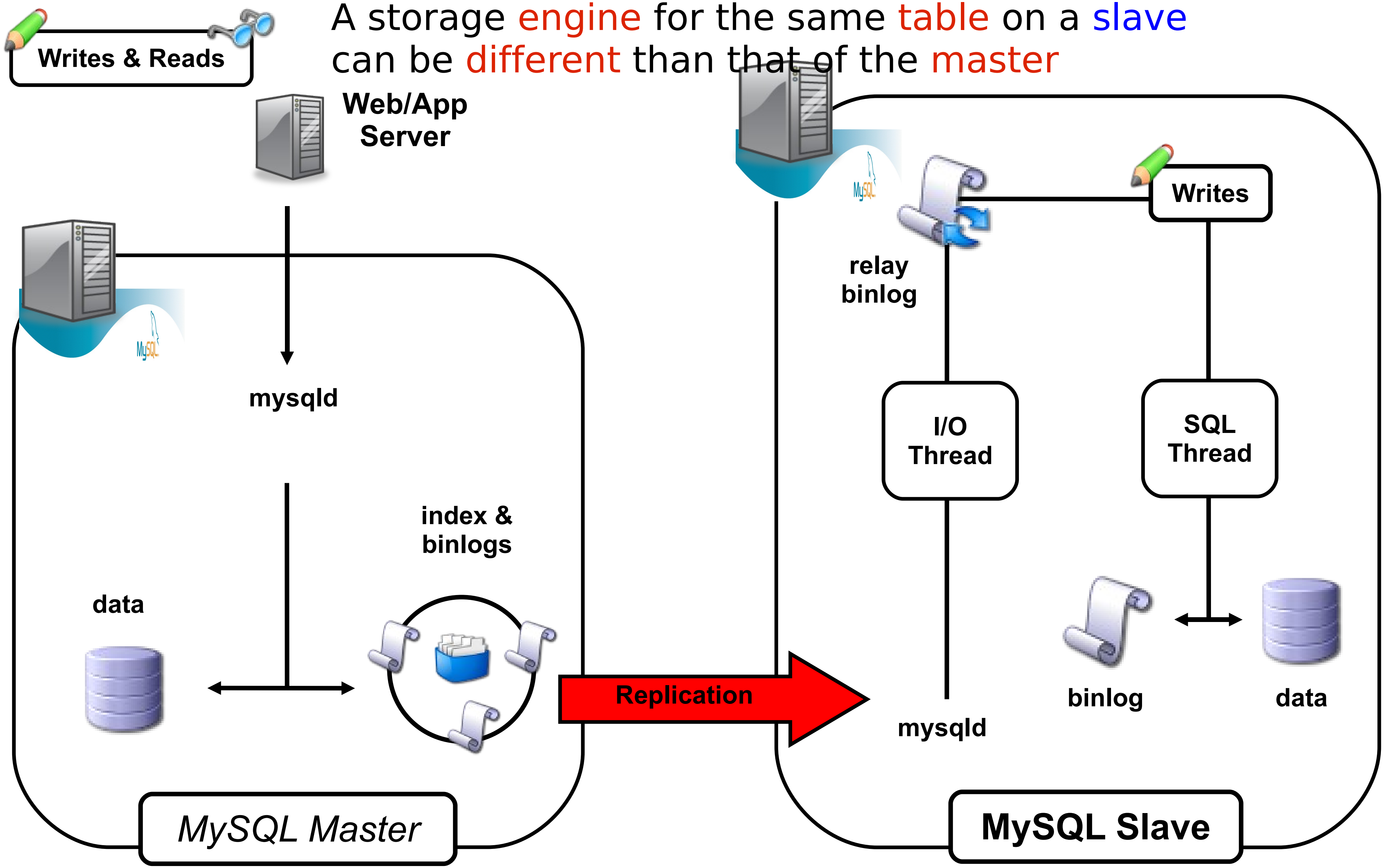
innodb

slave

isam

Inside MySQL Replication

A storage engine for the same table on a slave can be different than that of the master



Using different engines

Creating a table with a specified engine

```
CREATE TABLE t1 (...) ENGINE=InnoDB;
```

Changing existing tables

```
ALTER TABLE t1 ENGINE=MyISAM;
```

Finding all your available engines

```
SHOW STORAGE ENGINES;
```

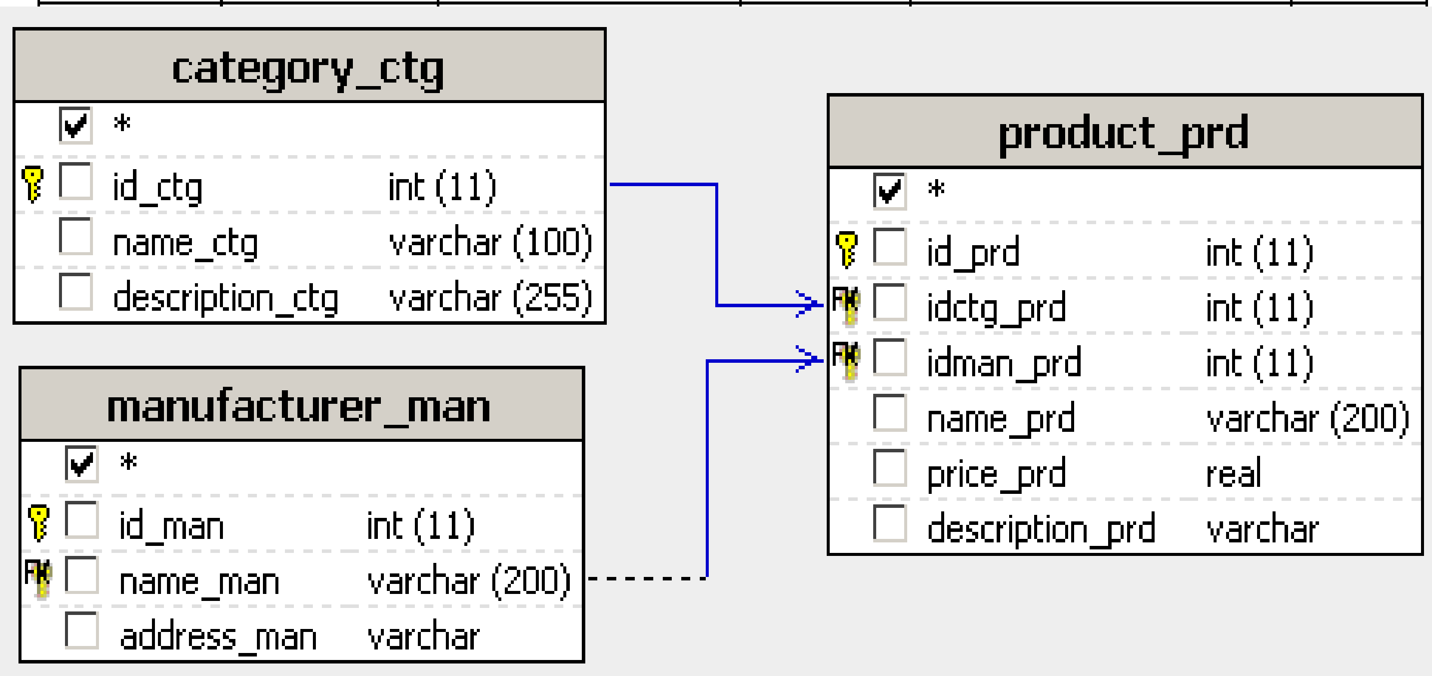
The schema

Basic foundation of performance

- Normalization
- Data Types
  - Smaller, smaller, smaller
    - Smaller tables use less disk, less memory, can give better performance
- Indexing
  - Speeds up retrieval

Goal of Normalization

Category	Manufacturer	Address	Product name	Description	Price
Electronics	ACME	15 Sunset BD, Los Angeles	TV Set	This high resolution plasma TV set will make a tempting choice for all viewers	$299.00
Media	ACME	15 Sunset BD, Los Angeles	CD Player	CD-R/RW compatible. Programmable. Optical digital output. Silver Colour.	$50.00
Cameras	CamProd	18, HW Street	QuickCam	QuickCam provides easy filming experience	$150.00



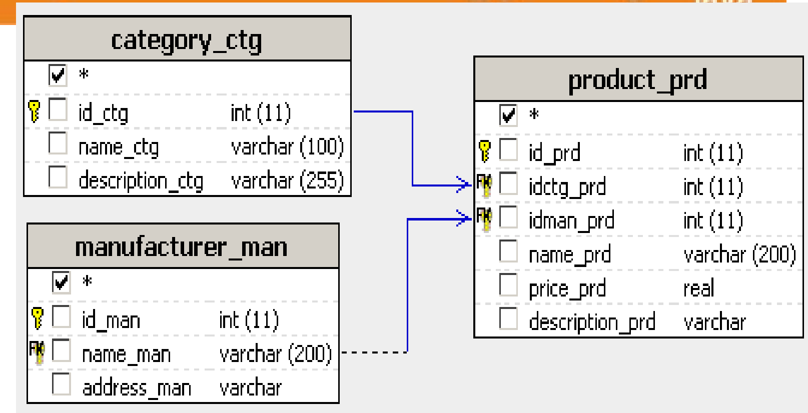
Eliminate redundant data:

Don't store the same data in more than one table

Only store related data in a table

reduces database size and errors

Normalization



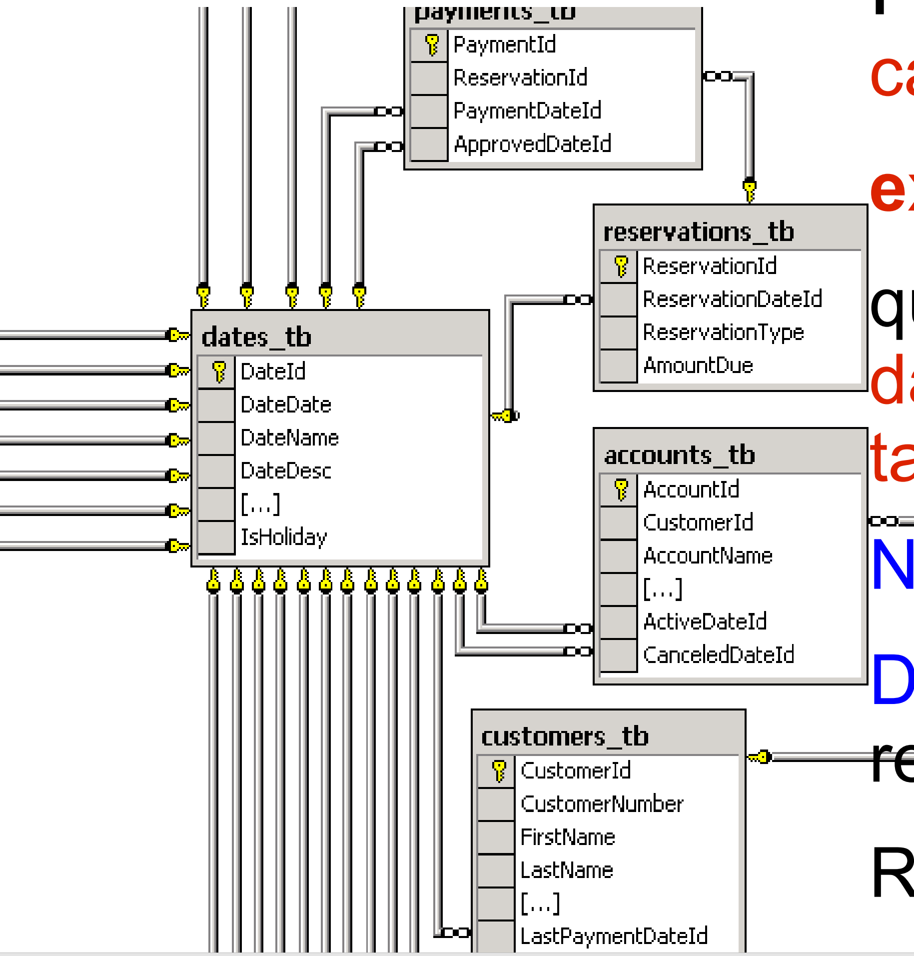
updates are usually faster.

there's less data to change.

tables are usually smaller, use less memory, which can give better performance.

better performance for distinct or group by queries

taking normalization *way* too far



However Normalized database causes joins for queries

excessively normalized database:

queries take more time to complete, as data has to be retrieved from more tables.

Normalized better for writes OLTP

De-normalized better for reads , reporting

Real World **Mixture**:

Normalize first
denormalize later

normalized schema

Cache selected columns in memory table

http://thedailywtf.com/forums/thread/75982.aspx

Data Types: Smaller, smaller, smaller

Smaller = less disk=less memory= better performance

Use the smallest data type possible

The **smaller** your data types,
The **more** index (and **data**) can fit into a block of **memory**, the **faster** your **queries** will be.

Period.

Especially for indexed fields

Choose your Numeric Data Type

MySQL has 9 numeric data types

Compared to Oracle's 1

Integer:

TINYINT , SMALLINT, MEDIUMINT, INT, BIGINT

Require 8, 16, 24, 32, and 64 bits of space.

Use UNSIGNED when you don't need negative numbers –

one more level of data integrity

BIGINT is **NOT** needed for AUTO_INCREMENT

INT UNSIGNED stores 4.3 billion values!

Choose your Numeric Data Type

Floating Point: FLOAT, DOUBLE

Approximate calculations

Fixed Point: DECIMAL

Always use DECIMAL for monetary/currency fields, never use FLOAT or DOUBLE!

Other: BIT

Store 0,1 values

Character Data Types

VARCHAR(n) variable length

uses only space it needs

Can save disk space = better performance

Use :

Max column length > avg

when updates rare (updates fragment)

CHAR(n) fixed length

Use:

short strings, Mostly same length, or changed frequently

Appropriate Data Types

Always define columns as NOT NULL

unless there is a good reason not to

Can save a byte per column

nullable columns make indexes, index statistics, and value comparisons more complicated.

Use the same data types for columns that will be compared in JOINs

Otherwise converted for comparison

smaller, smaller, smaller

The more records you can fit into a single page of memory/disk, the faster your seeks and scans will be

Use appropriate data types

Keep **primary keys small**

Use **TEXT** sparingly

Consider separate tables

Use **BLOBs** very sparingly

Use the filesystem for what it was intended

Indexes

Indexes Speed up Querys,

`SELECT...WHERE name = 'carol'`

only if there is good selectivity:

% of *distinct* values in a column

But... each index will slow down INSERT, UPDATE, and DELETE operations

Missing Indexes

Always have an **index** on join conditions

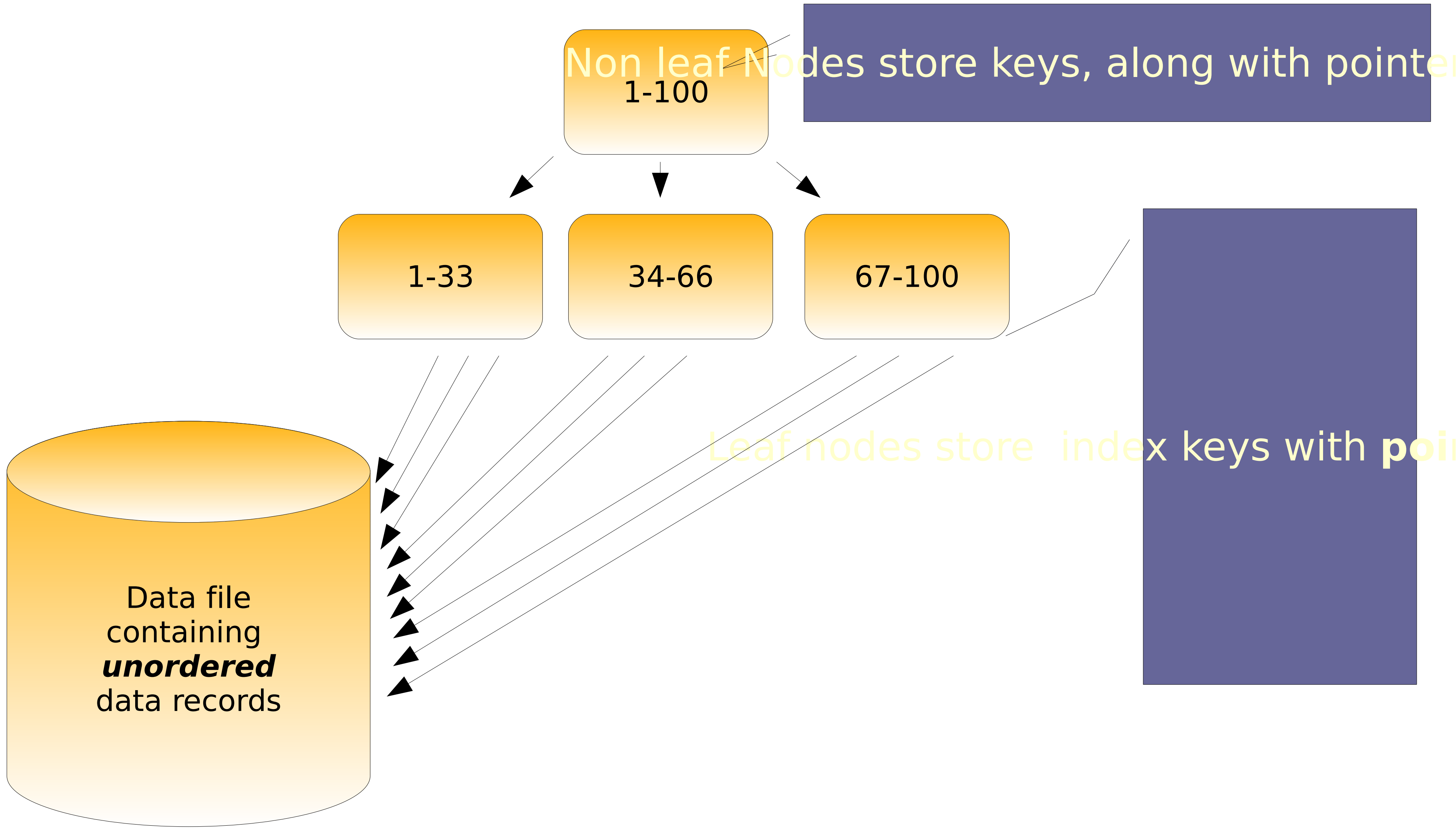
Look to add indexes on **columns** used in WHERE and **GROUP BY** expressions

`PRIMARY KEY`, `UNIQUE` , and `Foreign key Constraint` columns are automatically indexed.

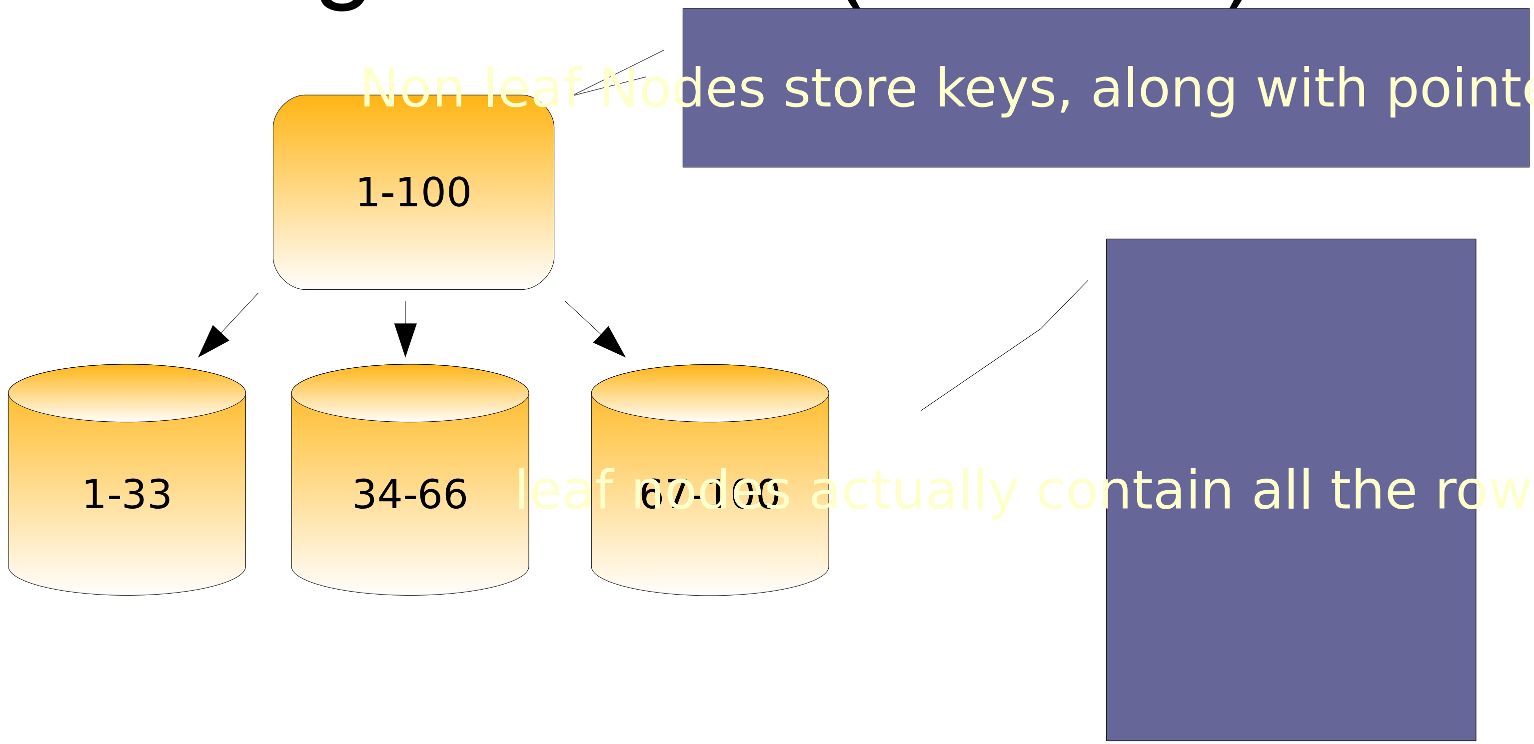
other columns can be indexed (`CREATE INDEX`..)

MyISAM index structure

Non-clustered organisation



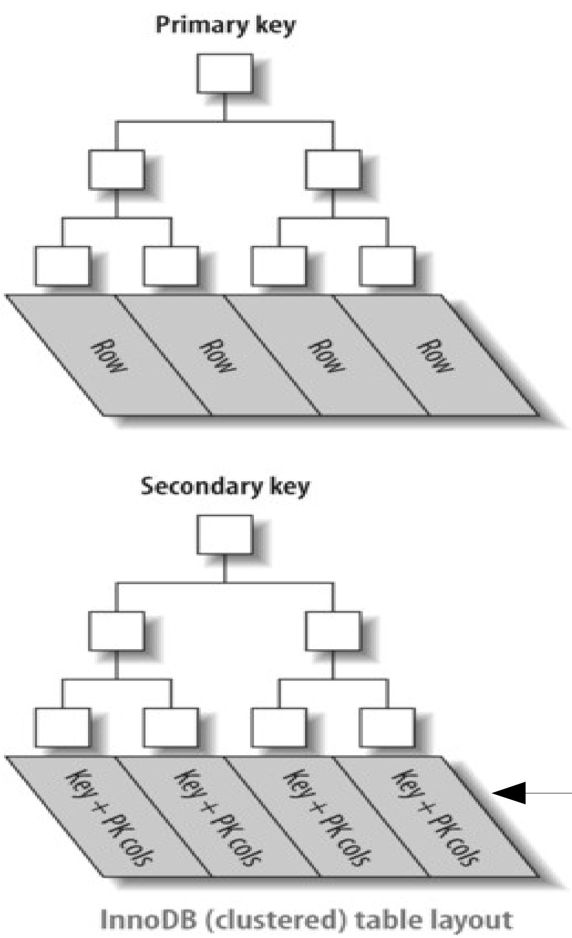
Clustered organisation (InnoDB)



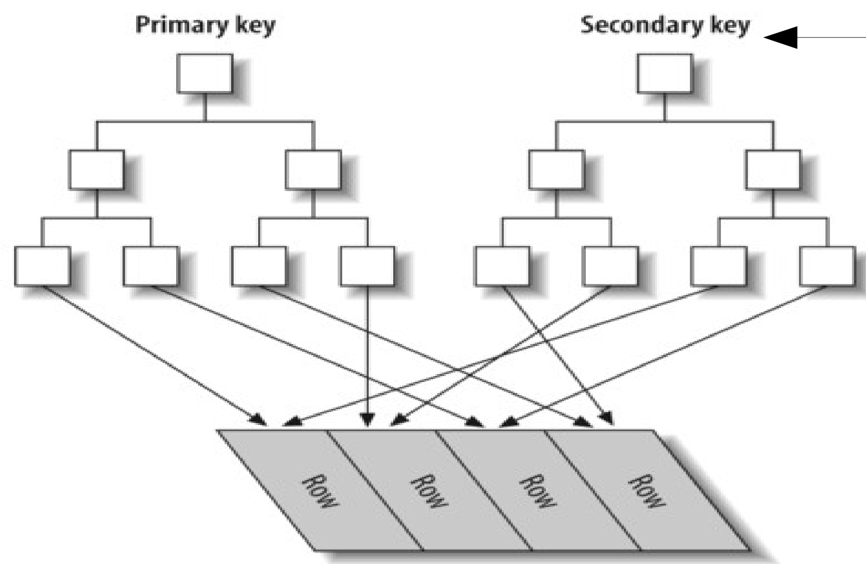
So, bottom line:

When looking up a record by a primary key, for a clustered layout/ organisation, the **lookup operation** (following the pointer from the leaf node to the data file) **is not needed.**

InnoDB (Clustered) indexes



InnoDB



ISAM

InnoDB: very important to have as small primary key as possible

Why? Primary key value is appended to every record in a secondary index

If you don't pick a primary key (bad idea!), one will be created for you

Ref: High Performance MySQL

B-tree indexes

B-Tree indexes work well for:

Match on key value

Match on range of values

avoid NULLS in the where clause - NULLS aren't indexed

Match on left most prefix

avoid LIKE beginning with %

Know how your Queries are executed by MySQL

harness the MySQL slow query log and use Explain

Append EXPLAIN to your SELECT statement

shows **how** the MySQL optimizer has chosen to execute the query

You Want to make your queries **access less** data:

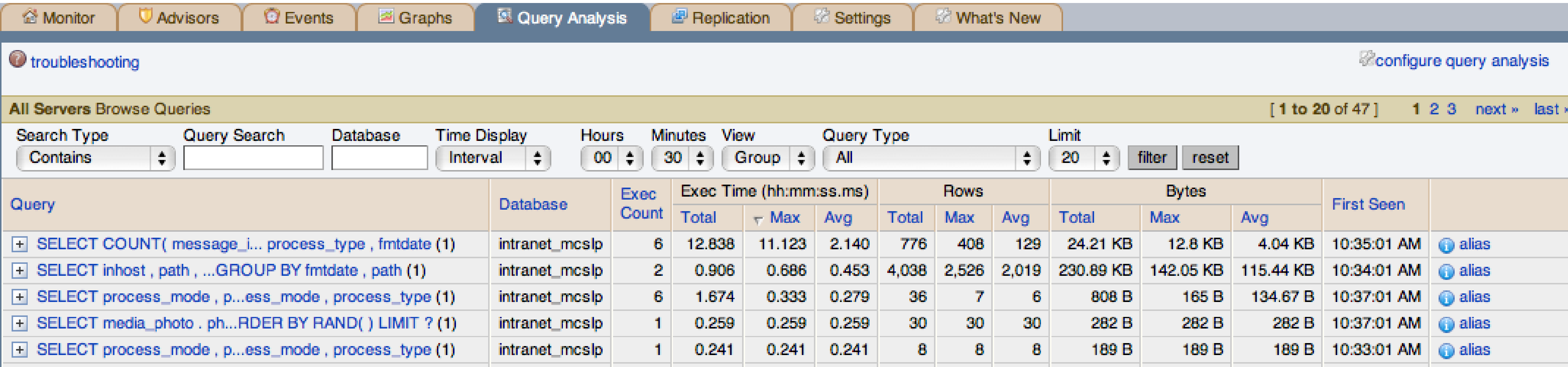
are queries accessing too many rows or columns?

Use to see where you should add indexes

Consider adding an index for slow queries

Helps **find missing indexes** early in the development process

MySQL Query Analyser


Monitor | Advisors | Events | Graphs | Query Analysis | Replication | Settings | What's New

troubleshooting | configure query analysis

All Servers Browse Queries [**1 to 20** of 47] **1** 2 3 next » last »

Search Type: Contains | Query Search | Database | Time Display: Interval | Hours: 00 | Minutes: 30 | View: Group | Query Type: All | Limit: 20 | filter | reset

Query	Database	Exec Count	Exec Time (hh:mm:ss.ms) Total	Max	Avg	Rows Total	Max	Avg	Bytes Total	Max	Avg	First Seen	
SELECT COUNT(message_i... process_type , fmtdate (1)	intranet_mcslp	6	12.838	11.123	2.140	776	408	129	24.21 KB	12.8 KB	4.04 KB	10:35:01 AM	alias
SELECT inhost , path , ...GROUP BY fmtdate , path (1)	intranet_mcslp	2	0.906	0.686	0.453	4,038	2,526	2,019	230.89 KB	142.05 KB	115.44 KB	10:34:01 AM	alias
SELECT process_mode , p...ess_mode , process_type (1)	intranet_mcslp	6	1.674	0.333	0.279	36	7	6	808 B	165 B	134.67 B	10:37:01 AM	alias
SELECT media_photo . ph...RDER BY RAND( ) LIMIT ? (1)	intranet_mcslp	1	0.259	0.259	0.259	30	30	30	282 B	282 B	282 B	10:37:01 AM	alias
SELECT process_mode , p...ess_mode , process_type (1)	intranet_mcslp	1	0.241	0.241	0.241	8	8	8	189 B	189 B	189 B	10:33:01 AM	alias

Find and fix problem SQL:

how long a query took

how the optimizer handled it

Drill downs, results of EXPLAIN statements

Historical and real-time analysis

query execution counts, run time

Its not just slow running queries that are a problem, Sometimes its SQL that **executes a lot** that kills your system

Understanding EXPLAIN

Just append EXPLAIN to your SELECT statement

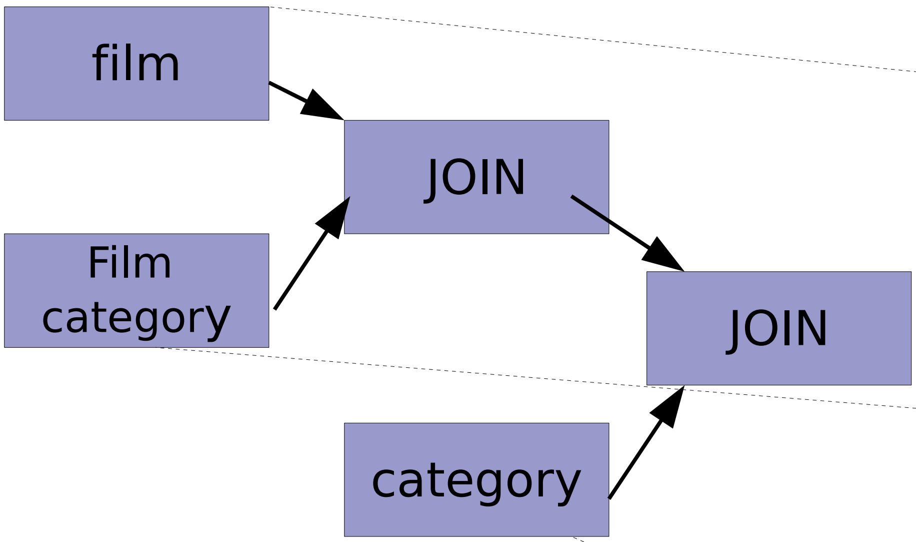
Provides the execution plan chosen by the MySQL optimizer for a specific SELECT statement

gives insight into how the MySQL optimizer has chosen to execute the query

Use to see where you should add indexes

ensures that missing indexes are picked up early in the development process

EXPLAIN: the execution plan




```
mysql> EXPLAIN SELECT f.film_id, f.title, c.name
    > FROM film f INNER JOIN film_category fc
    > ON f.film_id=fc.film_id INNER JOIN category c
    > ON fc.category_id=c.category_id WHERE f.title LIKE 'T%' \G
*************************** 1. row ***************************
  select_type: SIMPLE
        table: c
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16
        Extra:
*************************** 2. row ***************************
  select_type: SIMPLE
        table: fc
         type: ref
possible_keys: PRIMARY,fk_film_category_category
          key: fk_film_category_category
      key_len: 1
          ref: sakila.c.category_id
         rows: 1
        Extra: Using index
*************************** 3. row ***************************
  select_type: SIMPLE
        table: f
         type: eq_ref
possible_keys: PRIMARY,idx_title
          key: PRIMARY
      key_len: 2
          ref: sakila.fc.film_id
         rows: 1
        Extra: Using where
```

An *estimate* of rows in th
set

The “access stra
chosen

The ava
the

A covering index

EXPLAIN returns a **row of information** for each "**table**" used in the SELECT statement

The "table" can mean a real table, a temporary table, a subquery, a union result.

EXPLAIN example

```
mysql> EXPLAIN SELECT f.film_id, f.title, c.name
     > FROM film f INNER JOIN film_category fc
     > ON f.film_id=fc.film_id INNER JOIN category c
     > ON fc.category_id=c.category_id WHERE f.title LIKE 'T%' \G
*************************** 1. row ***************************
  select_type: SIMPLE
        table: c
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16
        Extra:
*************************** 2. row ***************************
  select_type: SIMPLE
        table: fc
         type: ref
possible_keys: PRIMARY,fk_film_category_category
          key: fk_film_category_category
      key_len: 1
          ref: sakila.c.category_id
         rows: 1
        Extra: Using index
*************************** 3. row ***************************
  select_type: SIMPLE
        table: f
         type: eq_ref
possible_keys: PRIMARY,idx_title
          key: PRIMARY
      key_len: 2
          ref: sakila.fc.film_id
         rows: 1
        Extra: Using where
```

Each row represents information used in SELECT

An *estimate* of rows in this set

The "access strategy" chosen

How MySQL will access the rows to find results

The available indexes, and the one(s) chosen

A **covering index** is used

Full Table Scan

```
EXPLAIN SELECT * FROM customer
           id: 1
  select_type: SIMPLE
        table: customer
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 2
        Extra: Using where
```

BAD
Using SELECT * FROM
No WHERE condition

type: shows the "access strategy"

full table scan

Avoid:
ensure indexes are on columns that are used in the WHERE, ON, and GROUP BY clauses.

Understanding EXPLAIN

```
EXPLAIN SELECT * FROM customer WHERE custid=1
           id: 1
  select_type: SIMPLE
        table: customer
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
        Extra:
```

constant

primary key lookup

primary key used in the WHERE

very fast because the table has at most one matching row

Range Access type

```
EXPLAIN SELECT * FROM rental WHERE rental_date
  BETWEEN '2005-06-14' AND '2005-06-16'

           id: 1
  select_type: SIMPLE
        table: rental
         type: range
possible_keys: rental_date
          key: rental_date
      key_len: 8
          ref: null
         rows: 364
        Extra: Using where
```

rental_date must be Indexed

Full Table Scan

```
EXPLAIN SELECT * FROM rental WHERE rental_date
  BETWEEN '2005-06-14' AND '2005-05-16'

           id: 1
  select_type: SIMPLE
        table: rental
         type: ALL
possible_keys: rental_date
          key: null
      key_len: null
          ref: null
         rows: 16000
        Extra: Using where
```

when range returns a lot of rows, > 20% table, forces scan

If too many rows estimated returned, scan will be used instead

Scans and seeks

A **seek** jumps to a place (on disk or in memory) to fetch row data

Repeat for each row of data needed

A **scan** will jump to the start of the data, and sequentially read (from either disk or memory) until the end of the data

Large amounts of data?

Scan operations are usually better than **many** seek operations

When optimizer sees a condition will return **> ~20%** of the rows in a table, it will prefer a scan versus many seeks

When do you get a full table scan?

No WHERE condition (duh.)

No index on any field in WHERE condition

Poor selectivity on an indexed field

Too many records meet WHERE condition

scans can be a sign of poor indexing

Covering indexes

When all columns needed from a single table for a SELECT are available in the index

No need to grab the rest of the columns from the data (file or page)

Shows up in Extra column of EXPLAIN as "Using index"

Important to know the data index organisation of the storage engine!

Understanding EXPLAIN

There is a huge difference between “index” in the type column and “Using index” in the Extra column

type column: "access strategy"

Const: primary key= good

ref: index access =good

index: index tree is scanned = bad

ALL: A full table scan = bad

means a full **index tree scan** (**bad!**)

Extra column: additional information

Using index=good

filesort or Using temporary = bad

means a **covering index** was found (good**!**)

EXPLAIN example

```
mysql> EXPLAIN SELECT f.film_id, f.title, c.name
     > FROM film f INNER JOIN film_category fc
     > ON f.film_id=fc.film_id INNER JOIN category c
     > ON fc.category_id=c.category_id WHERE f.title LIKE 'T%' \G
*************************** 1. row ***************************
  select_type: SIMPLE
        table: c
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16
        Extra:
*************************** 2. row ***************************
  select_type: SIMPLE
        table: fc
         type: ref
possible_keys: PRIMARY,fk_film_category_category
          key: fk_film_category_category
      key_len: 1
          ref: sakila.c.category_id
         rows: 1
        Extra: Using index
*************************** 3. row ***************************
  select_type: SIMPLE
        table: f
         type: eq_ref
possible_keys: PRIMARY,idx_title
          key: PRIMARY
      key_len: 2
          ref: sakila.fc.film_id
         rows: 1
        Extra: Using where
```

An *estimate* of rows in this set

The "access strategy" chosen

The available indexes, and the one(s) chosen

A **covering index** is used

Covering indexes are useful. Why? Query execution fully from index, without having to read the data row!

Operating on indexed column with a function

Indexes speed up SELECTs on a column, but...

indexed column within a function cannot be used

SELECT ... WHERE SUBSTR(name, 3)

Most of the time, there are ways to rewrite the query to isolate the indexed column on left side of the equation

Indexed columns and functions don't mix

indexed column should be alone on left of comparison

```
mysql> EXPLAIN SELECT * FROM film WHERE title LIKE 'Tr%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: range
possible_keys: idx_title
          key: idx_title
      key_len: 767
          ref: NULL
         rows: 15
        Extra: Using where
```

Nice. In the top query, we have a fast range a

```
mysql> EXPLAIN SELECT * FROM film WHERE LEFT(title,2) = 'Tr' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra: Using where
```

Oops. here we have a slower full table sca

Partitioning

Vertical partitioning

Split tables with many columns into multiple tables

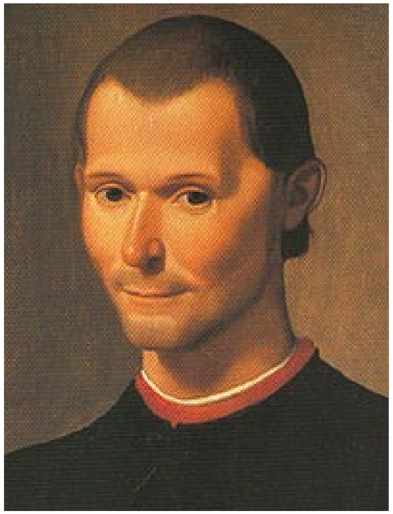
limit number of columns per table

Horizontal partitioning

Split table by rows into partitions

Both are important for different reasons

Partitioning in MySQL 5.1 is *horizontal partitioning for data warehousing*



Niccolò Machiavelli
The Art of War, (1519-1520): ***divide the forces of the enemy***

vertical partitioning

```
CREATE TABLE Users (
  user_id INT NOT NULL
AUTO_INCREMENT
, email VARCHAR(80) NOT NULL
, display_name VARCHAR(50) NOT
NULL
, password CHAR(41) NOT NULL
, first_name VARCHAR(25) NOT NULL
, last_name VARCHAR(25) NOT NULL
, address VARCHAR(80) NOT NULL
, city VARCHAR(30) NOT NULL
, province CHAR(2) NOT NULL
, postcode CHAR(7) NOT NULL
, interests TEXT NULL
, bio TEXT NULL
, signature TEXT NULL
, skills TEXT NULL
, PRIMARY KEY (user_id)
, UNIQUE INDEX (email)
) ENGINE=InnoDB;
```

Frequently referenced

```
CREATE TABLE Users (
  user_id INT NOT NULL AUTO_INCREMENT
, email VARCHAR(80) NOT NULL
, display_name VARCHAR(50) NOT NULL
, password CHAR(41) NOT NULL
, PRIMARY KEY (user_id)
, UNIQUE INDEX (email)
) ENGINE=InnoDB;
```

Less Frequently referenced, TEXT data

```
CREATE TABLE UserExtra (
  user_id INT NOT NULL
, first_name VARCHAR(25) NOT NULL
, last_name VARCHAR(25) NOT NULL
, address VARCHAR(80) NOT NULL
, city VARCHAR(30) NOT NULL
, province CHAR(2) NOT NULL
, postcode CHAR(7) NOT NULL
, interests TEXT NULL
, bio TEXT NULL
, signature TEXT NULL
, skills TEXT NULL
, PRIMARY KEY (user_id)
, FULLTEXT KEY (interests, skills)
) ENGINE=MyISAM;
```

Mixing frequently and infrequently accessed attributes in a single table?

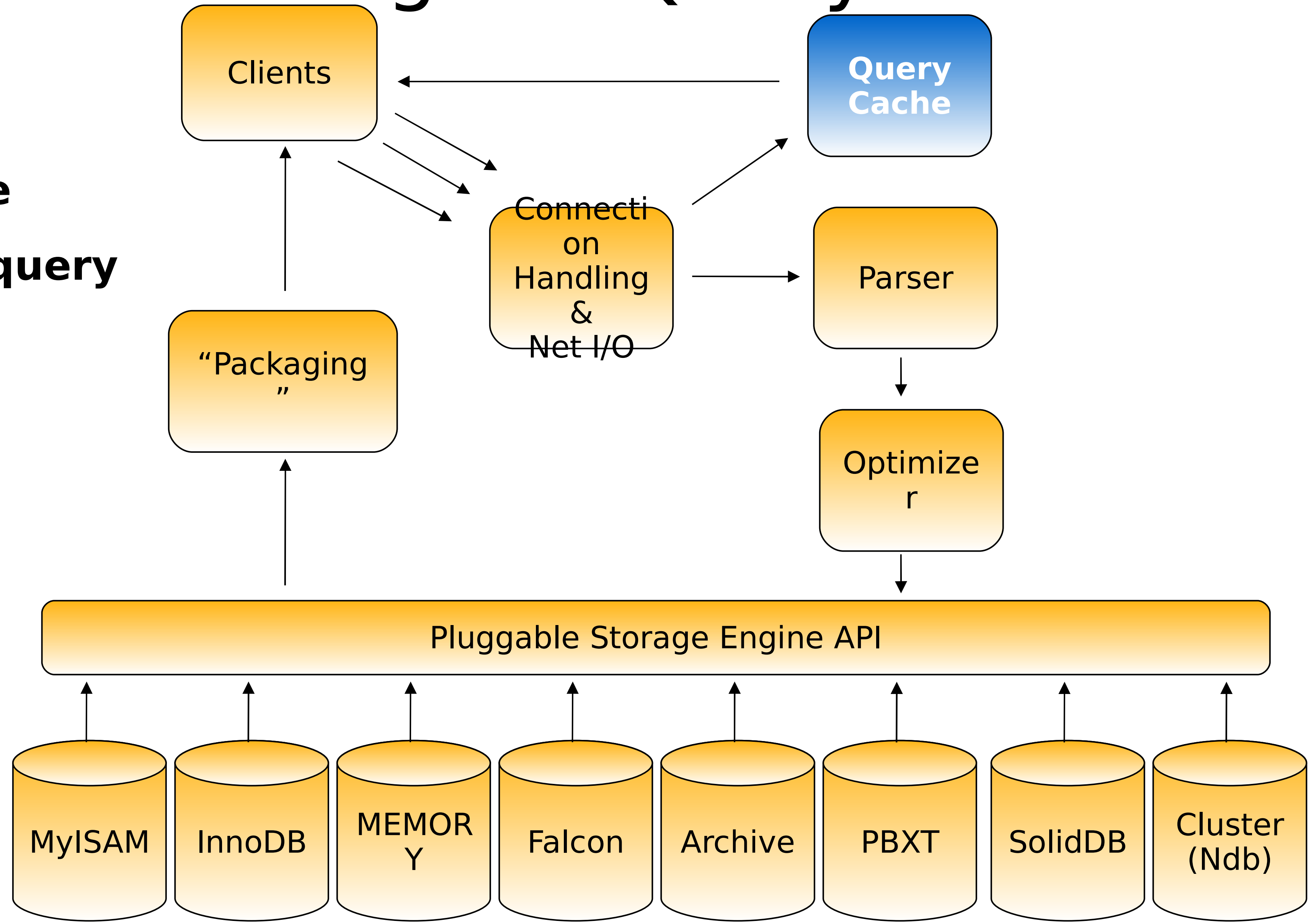
Space in buffer pool at a premium?

Splitting the table allows main records to consume the buffer pages without the extra data taking up space in memory

Need **FULLTEXT** on your text columns?

Understanding the Query Cache

Caches the complete query



Query cache

Caches the complete query

Coarse invalidation

any modification to ***any*** table in the SELECT invalidates ***any*** cache entry which uses that table

Good for read mostly tables

Fast query when no table changes

Remedy with vertical table partitioning

vertical partitioning ... continued

```
CREATE TABLE Products (
  product_id INT NOT NULL
, name VARCHAR(80) NOT NULL
, unit_cost DECIMAL(7,2) NOT NULL
, description TEXT NULL
, image_path TEXT NULL
, num_views INT UNSIGNED NOT NULL
, num_in_stock INT UNSIGNED NOT NULL
, num_on_order INT UNSIGNED NOT NULL
, PRIMARY KEY (product_id)
, INDEX (name(20))
) ENGINE=InnoDB;

// Getting a simple COUNT of products
// easy on MyISAM, terrible on InnoDB
SELECT COUNT(*)
FROM Products;
```

static attributes

```
CREATE TABLE Products (
  product_id INT NOT NULL
, name VARCHAR(80) NOT NULL
, unit_cost DECIMAL(7,2) NOT NULL
, description TEXT NULL
, image_path TEXT NULL
, PRIMARY KEY (product_id)
, INDEX (name(20))
) ENGINE=InnoDB;
```

frequently updated fields

```
CREATE TABLE ProductCounts (
  product_id INT NOT NULL
, num_views INT UNSIGNED NOT NULL
, num_in_stock INT UNSIGNED NOT NULL
, num_on_order INT UNSIGNED NOT NULL
, PRIMARY KEY (product_id)
) ENGINE=InnoDB;
```

count

```
CREATE TABLE TableCounts (
  total_products INT UNSIGNED NOT NULL
) ENGINE=MEMORY;
```

Mixing static attributes with frequently updated fields in a single table?

Each time an update occurs, queries referencing the table invalidated in the query cache

Doing **COUNT(*)** with no **WHERE** on an indexed field on an InnoDB table?

full table counts slow InnoDB table

Solving multiple problems in one query

We want to get the orders that were created in the last 7 days

```
SELECT * FROM Orders WHERE TO_DAYS(CURRENT_DATE())
– TO_DAYS(order_created) <= 7;
```

First, we are operating on an **indexed column** (**order_created**) with a **function TO_DAYS**– let's **fix** that:

```
SELECT * FROM Orders WHERE order_created >=
CURRENT_DATE() - INTERVAL 7 DAY;
```

we rewrote the **WHERE** expression to remove the function on the index, we still have a the function **CURRENT_DATE()**in the statement, which eliminates this query from being placed in the query cache – let's fix that

Solving multiple problems in one query

– let's fix that:

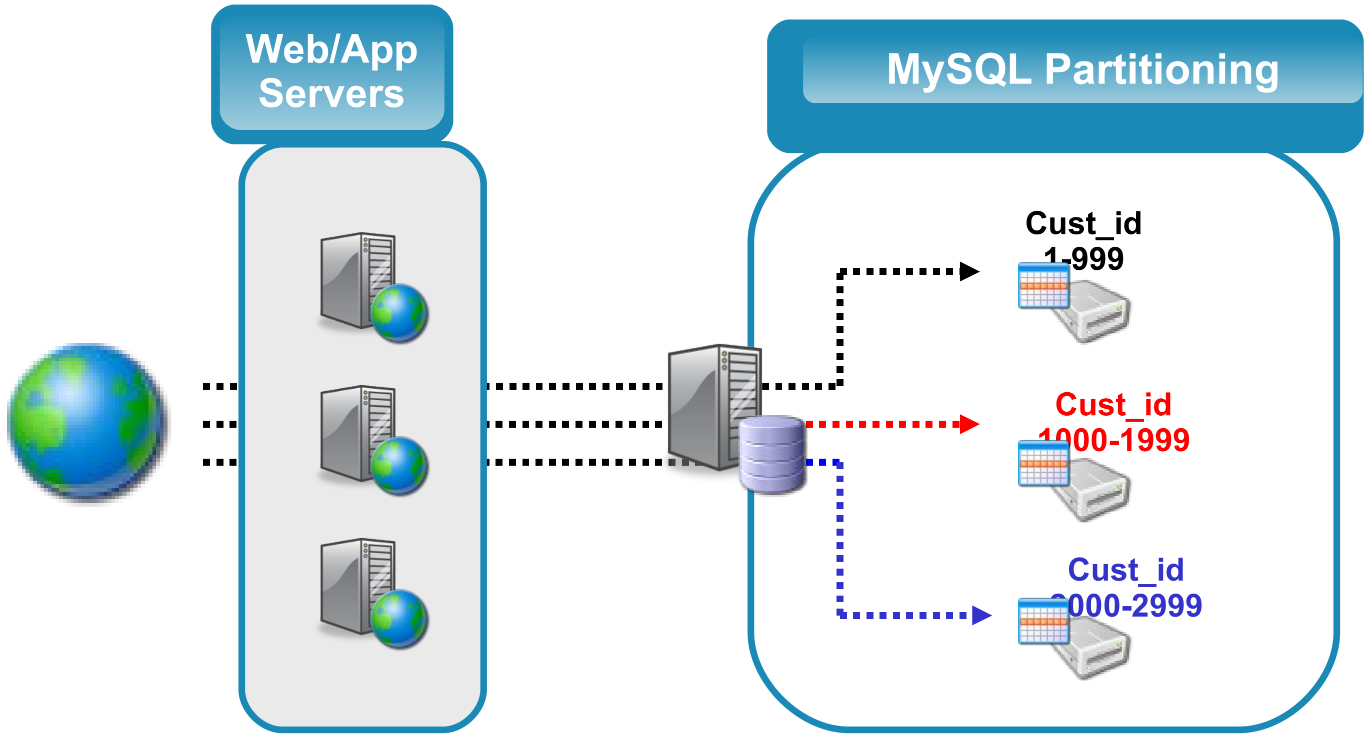
```
SELECT * FROM Orders WHERE order_created >= '2008-01-11'
- INTERVAL 7 DAY;
```

We replaced the function **CURRENT_DATE()** with a **constant**. However, we are specifying **SELECT *** instead of the actual fields we need from the table.

What if there are fields we **don't need**? Could cause **large result set** which **may not fit** into the **query cache** and may force a disk-based **temporary table**

```
SELECT order_id, customer_id, order_total,
order_created
FROM Orders WHERE order_created >= '2008-01-11' -
INTERVAL 7 DAY;
```

Scalability: MySQL 5.1 Horizontal Partitioning



CREATE TABLE cust (id int)
ENGINE=MyISAM

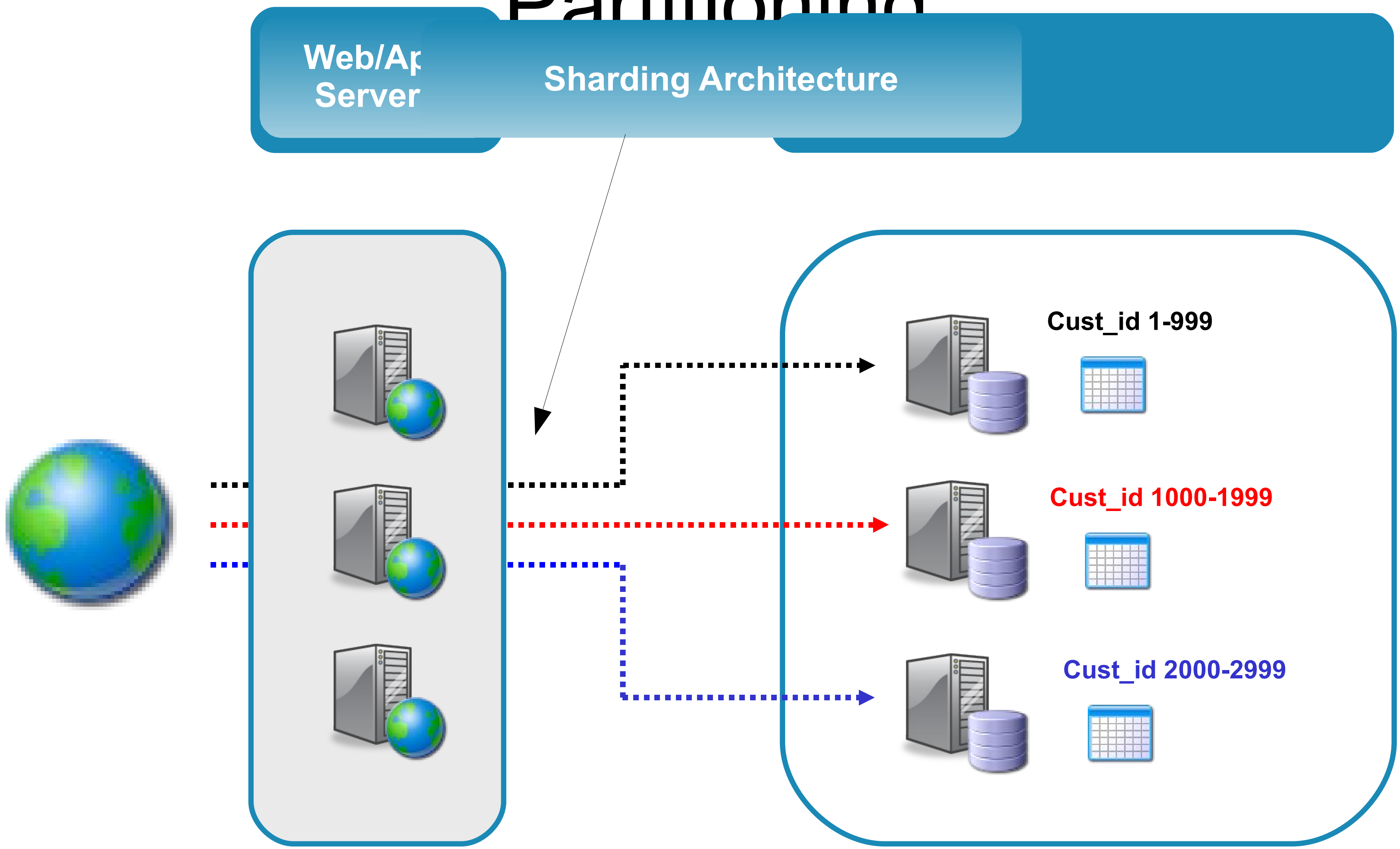
PARTITION BY RANGE (id)
(
PARTITION P1 VALUES LESS THAN (10),
PARTITION P2 VALUES LESS THAN (20)
)

Split table with many rows into partitions by range, key

Logical splitting of tables
 No need to create separate tables
• Transparent to user

Why?
•To make **range selects** faster
Good for
Data Warehouses
Archival and Date based partitioning

Scalability: Sharding - Application Partitioning



Lazy loading and JPA

```
public class Employee{
    @OneToMany(mappedBy = "employee")
    private Collection<Address> addresses; .....
}
```

Default FetchType is LAZY for 1:m and m:n relationships

benefits large objects and relationships

However for use cases where data is needed can cause n+1 selects

Capture generated SQL

persistence.xml file:<property name="toplink.logging.level" value="FINE">

examine the SQL statements

optimise the number of SQL statements executed!

only retrieve the data your application needs!

Lazy loading and JPA

```
public class Employee{

     @OneToMany(mappedBy = "employee",fetch = FetchType.EAGER)
     private Collection<Address> addresses;

  .....
 }
```

Relationship can be Loaded Eagerly

if you have several related relationships, could load too much !

OR

Temporarily override the LAZY fetch type, use Fetch Join in a query:

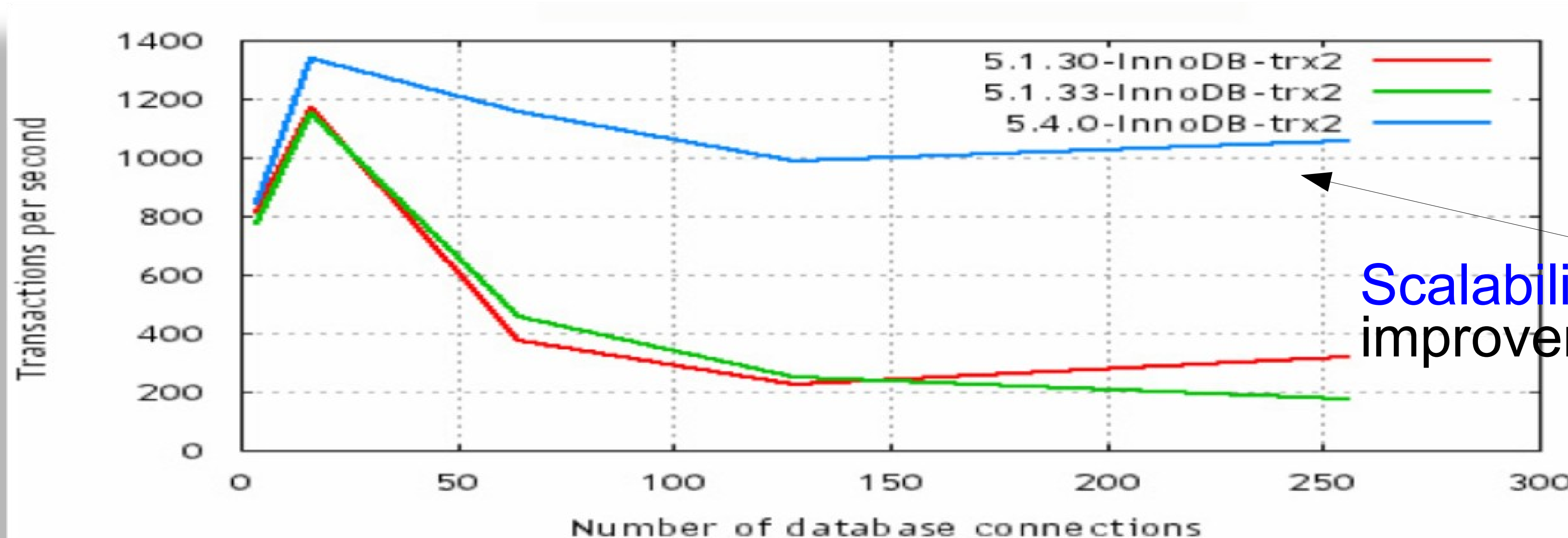
```
@NamedQueries({ @NamedQuery(name="getItEarly",
        query="SELECT e FROM Employee e JOIN FETCH e.addresses")})

public class Employee{
.....
}
```

MySQL Server 5.4

Scalability improvements -
more CPU’s / cores than before.
MySQL/InnoDB scales up to 16-way x86 servers and 64-way CMT server
Subquery optimizations
decrease response times (in some cases > 99%)
New join methods
improve speed of queries
And more (**Dtrace** probes, replication heartbeat)…
GA Target: December 2009

MySQL Server 5.4

Solaris x86 sysbench benchmark – MySQL 5.4 vs. 5.1

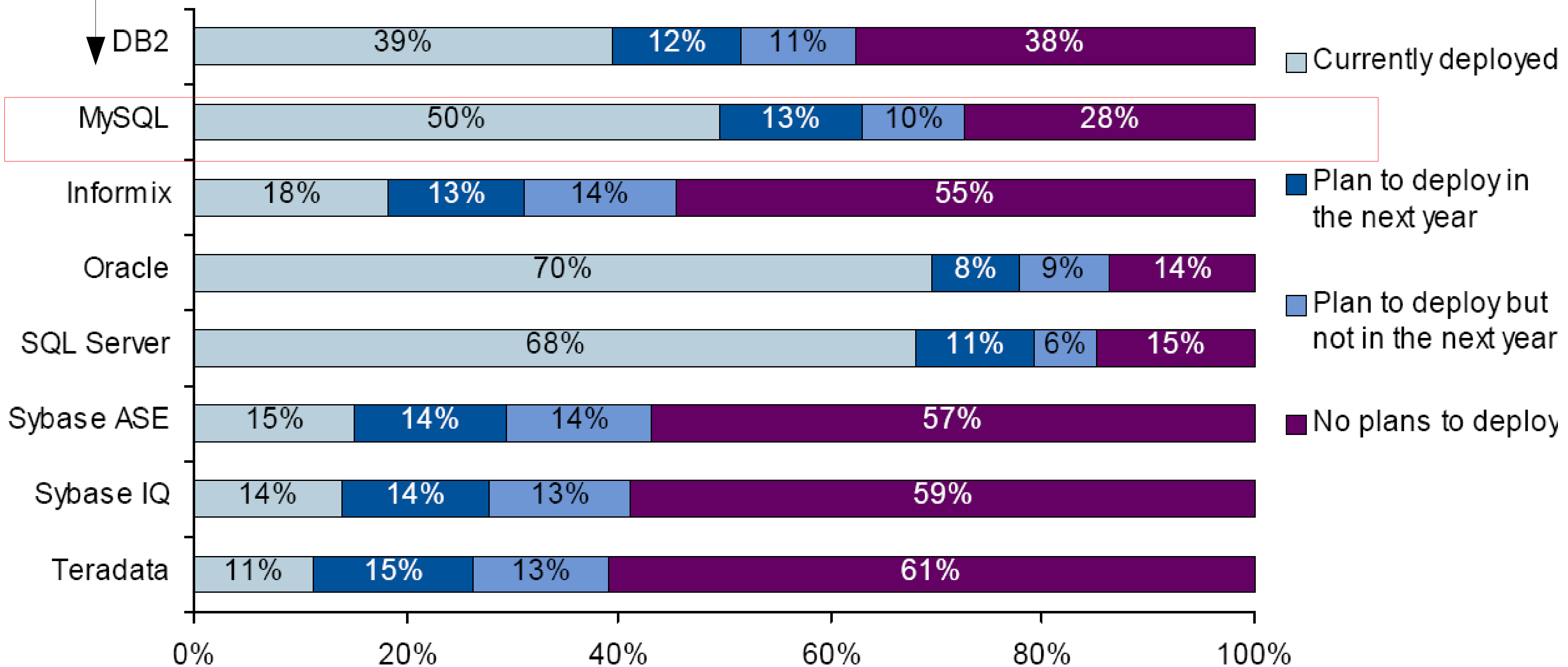


Scalability improvements

Requirements	MySQL Replica	MySQL Replication + Hea	MySQL, Heartbeat + DRB	MySQL Cluster
Availability				
Automated IP Fail Over	No	Yes	Yes	No
Automated DB Fail Over	No	No	Yes	Yes
Typical Fail Over Time	Varies	Varies	< 30 secs	< 3 secs
Auto Resynch of Data	No	No	Yes	Yes
Geographic Redundancy	Yes	Yes	MySQL Replication	MySQL Replication
Scalability				
Built-in Load Balancing	MySQL Replication	MySQL Replication	MySQL Replication	Yes
Read Intensive	Yes	Yes	MySQL Replication	Yes
Write Intensive	No	No	If configured correctly	Yes
# of Nodes per Cluster	Master/Slave(s)	Master/Slave(s)	Active/Passive	255
# of Slaves	Dozens for Reads	Dozens for Reads	Dozens for Reads	Dozens for Reads

MySQL: #3 Most Deployed Database

63% Are Deploying MySQL or Are Planning To Deploy



Gartner

MySQL Enterprise

Server

MySQL Enterprise Server
Monthly Rapid Updates
Quarterly Service Packs
Hot Fix Program
Extended End-of-Life

Monitor

Global Monitoring of All Servers
Web-Based Central Console
Built-in Advisors
Expert Advice
Specialized Scale-Out Help

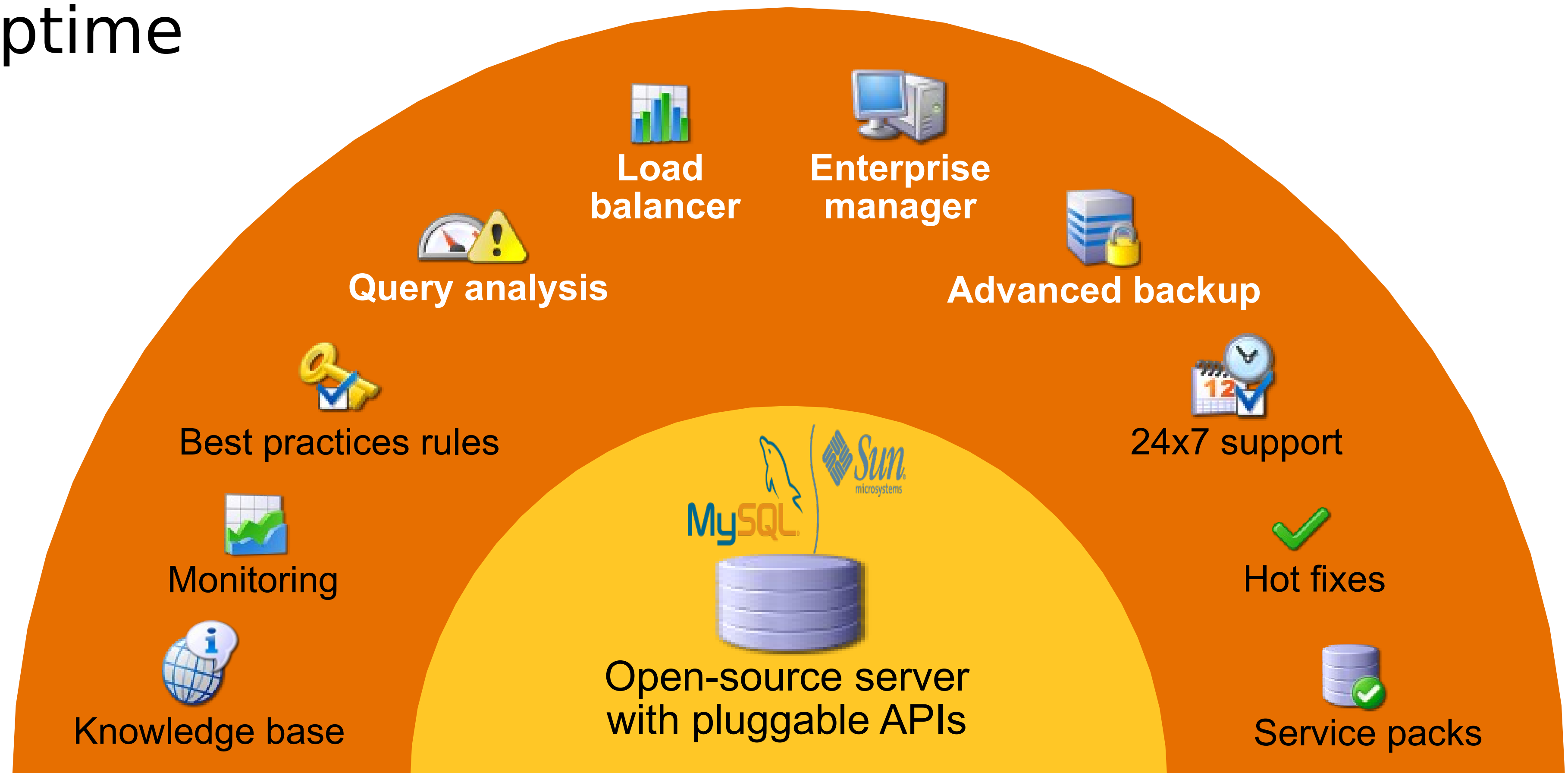
Support

24 x 7 x 365 Production Support
Web-Based Knowledge Base
Consultative Help
Bug Escalation Program

Added Value of MySQL Enterprise

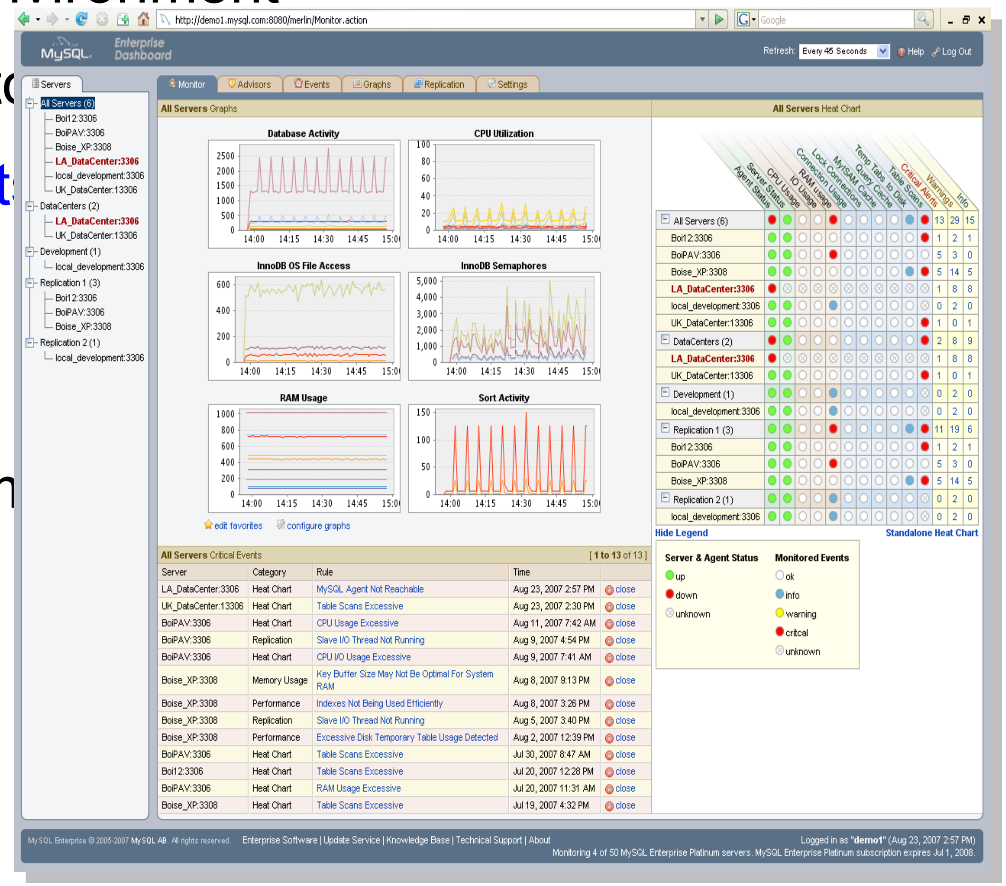
Comprehensive offering of production support, monitoring tools, and MySQL database software

Optimal performance, reliability, security, and uptime



MySQL Enterprise Monitor

- consolidated view into entire MySQL environment
- scovery of MySQL servers, replication to
- izable rules-based monitoring and alert
- s problems *before* they occur
- s risk of downtime
- t easier to scale out without requiring m



A Virtual MySQL DBA Assistant!


MYSQL
CASE STUDIES

facebook

Application

Facebook is a social networking site

Key Business Benefit

MySQL has enabled facebook to grow to 70 million users.

Why MySQL?

“We are one of the largest MySQL web sites in production. MySQL has been a revolution for young entrepreneurs.”

Owen Van Natta
Chief Operating Officer
Facebook

eBay

Application

Real-time personalization for advertising

Key Business Benefits

Handles eBay's personalization needs.

Manages 4 billion requests per day

Why MySQL Enterprise?

Cost-effective

Performance: 13,000 TPS on Sun Fire x4100

Scalability: Designed for 10x future growth

Monitoring: MySQL Enterprise Monitor

Chris Kasten,

Kernel Framework Group, eBay

Zappos

Application

$800 Million Online Retailer of shoes. Zappos stocks over 3 million items.

Key Business Benefit

Zappos selected MySQL because it was the most robust, affordable database software available at the time.

Why MySQL?

"MySQL provides the perfect blend of an enterprise-level database and a cost-effective technology solution. In my opinion, MySQL is the only database we would ever trust to power the Zappos.com website."

Kris Ongbongan,

IT Manager

Java Technology Day

Glassfish and MySQL Part 2

http://java.sun.com/developer/technicalArticles/glassfish/GFandMySQL_Part2.html

Most Visited Getting Started Latest Headlines Apple Yahoo! Google Maps YouTube Wikipedia News Popular

Sun Java Solaris Communities My SDN Account Join SDN

Sun Developer Network (SDN)

APIs Downloads Products Support Training Participate

» search tips Search

SDN Home > Java Technology > Reference > Technical Articles and Tips >

Article

GlassFish and MySQL, Part 2: Building a CRUD Web Application With Data Persistence

Print-friendly Version

By Ed Ort and Carol McDonald, November 2008

Article Index

This is the second article in a series of articles on GlassFish and MySQL. Part 1 of the series describes the advantages of using GlassFish with MySQL and illustrates why the combination is a perfect choice for developing and deploying web applications. In Part 2, you'll learn how to develop a create, read, update, delete (CRUD) web application that uses GlassFish and MySQL. The application uses the Java Persistence API implemented in GlassFish to manage data persistence.

An important characteristic of both GlassFish and MySQL is that they're easily integrated into popular development tools. For example, plug-ins are available for both GlassFish and MySQL to integrate them into the NetBeans IDE and the Eclipse IDE. In addition, NetBeans IDE 6.1 With GlassFish and MySQL Bundle Download is available that integrates GlassFish v2 Update Release 2 (UR2) and MySQL 5.0 Community Server into NetBeans IDE 6.1. You can also download GlassFish v2UR2 with either NetBeans IDE 6.1 or NetBeans IDE 6.5 in a single bundle. A precursor to the next version of GlassFish, called the GlassFish v3 Prelude, is also available with NetBeans IDE 6.5.

This article shows you how to use the NetBeans IDE with GlassFish and MySQL to create the CRUD application. Specifically, you'll take advantage of features in NetBeans IDE 6.5, GlassFish v2UR2, and MySQL 5.1 Community Server to build and deploy the application.

You can examine the completed CRUD application by downloading and expanding the petcatalog application package.

Contents

The Application

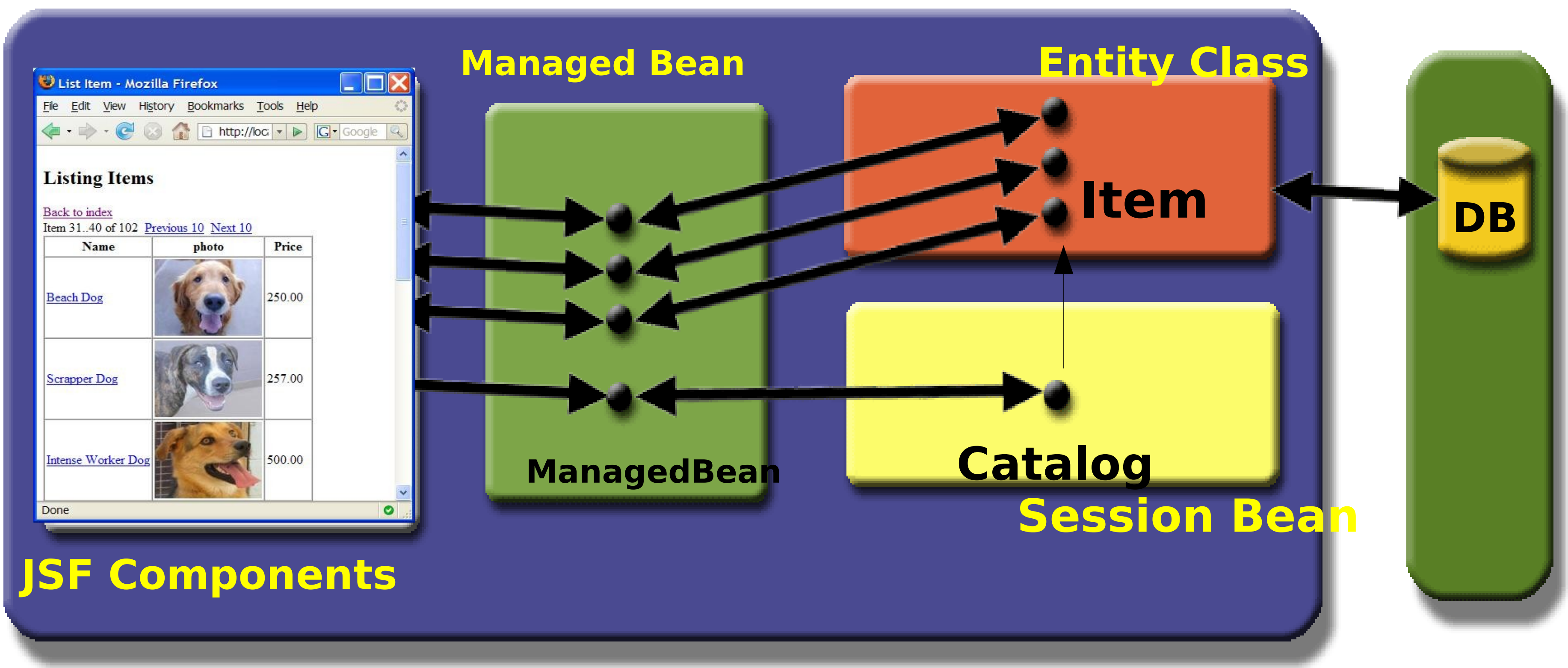
The application for this article allows users to search an online catalog of pets. For example, users can search for a specific type of pet, such as medium-sized dogs, and display information about the items of that type in the catalog. Figure 1 shows a page that the application displays with this type of information.

Ed Ort is a writer on the staff of the Sun Developer Network. He has written extensively about a wide variety of programming topics including relational database technology, programming languages, web services, and Ajax. Read his blog.

Carol McDonald is a Java technology evangelist at Sun. Her prolific blogging and wide-ranging programming skills make her a popular speaker at conferences such as Sun Tech Days.

FEEDBACK

Catalog Sample Java EE Application



Glassfish and MySQL Part 3

http://java.sun.com/developer/technicalArticles/glassfish/GFandMySQL_Part3.html

glassfish mysql part 3

Most Visited Getting Started Latest Headlines Apple Yahoo! Google Maps YouTube Wikipedia News Popular

Sun Java Solaris Communities My SDN Account Join SDN

Sun Developer Network (SDN)

APIs Downloads Products Support Training Participate

» search tips Search

SDN Home > Java Technology > Reference > Technical Articles and Tips >

Article

GlassFish and MySQL, Part 3: Creating and Using a Web Service

Print-friendly Version

By Ed Ort and Carol McDonald, January 2009

Articles Index

This is the third article in a series of articles on GlassFish and MySQL. Part 1 of the series describes the advantages of using GlassFish with MySQL and illustrates why the combination is a perfect choice for developing and deploying web applications.

Part 2 shows how to develop a create, read, update, and delete (CRUD) web application that uses GlassFish and MySQL. The application uses the Java Persistence API implemented in GlassFish to manage data persistence.

In Part 3, you'll learn how easy it is to convert the controller layer of the web application, that is, the layer of the application that performs the CRUD operations -- into a web service. You'll also learn how to create a client for the web service. As was the case for the web application discussed in Part 2, the web service discussed in Part 3 uses GlassFish, MySQL, and the Java Persistence API.

This article shows you how to use the NetBeans IDE with GlassFish and MySQL to create the web service and client. Specifically, you'll take advantage of features in NetBeans IDE 6.5, GlassFish v2UR2, and MySQL 5.1 Community Server to build and deploy the web service and client.

> This article shows you how to use the NetBeans IDE with GlassFish and MySQL to create a web service and client.

If you're not familiar with web services, see the Web Services section. Otherwise, skip to The Updated Application section. You can examine the completed web service and client by downloading and expanding the `CatalogService` and `CatalogClient` packages.

Contents

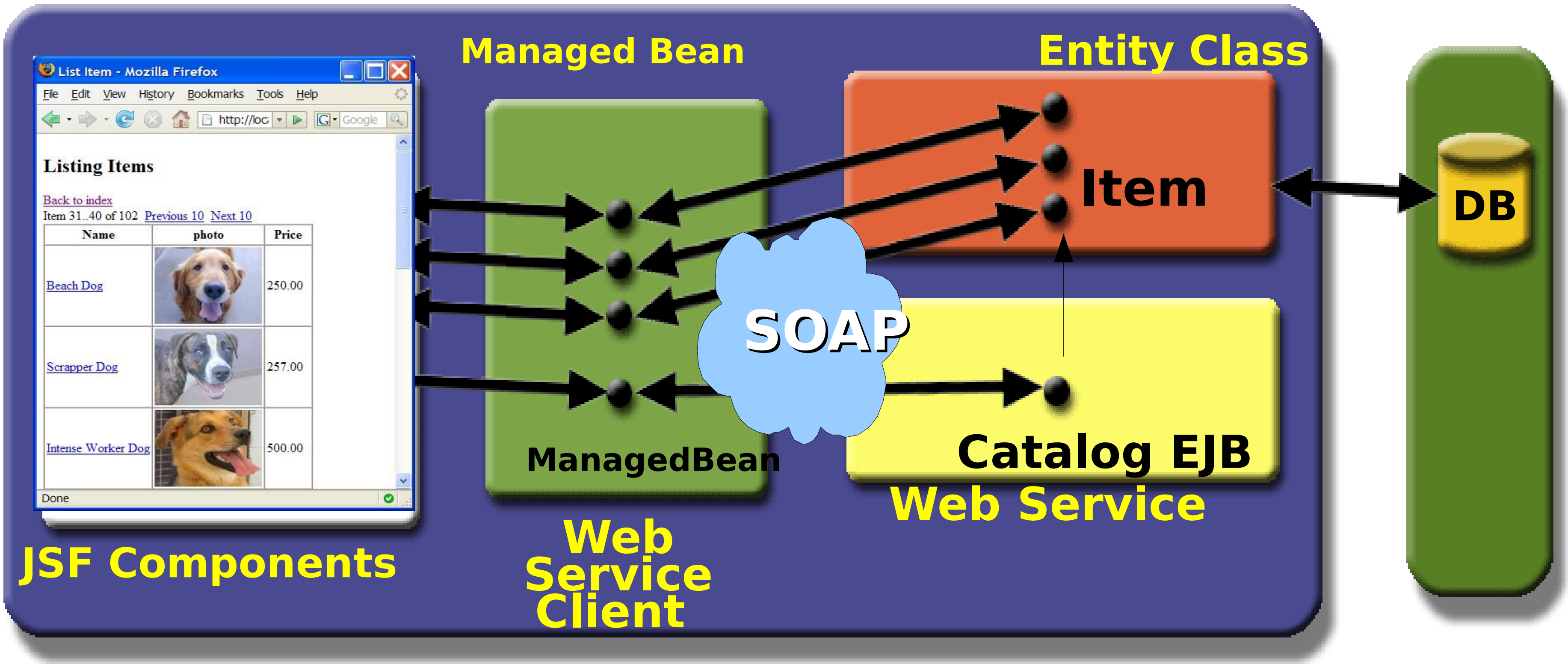
Ed Ort is a writer on the staff of the Sun Developer Network. He has written extensively about a wide variety of programming topics including relational database technology, programming languages, web services, and Ajax. Read his blog.

Carol McDonald is a Java technology evangelist at Sun. Her prolific blogging and wide-ranging programming skills make her a popular speaker at conferences such as Sun Tech Days.

FEEDBACK

Done

Catalog Sample JAX-WS Application



Glassfish and MySQL Part 4

GlassFish and MySQL, Part 4: Creating a RESTful Web Service and JavaFX Client - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://java.sun.com/developer/technicalArticles/glassfish/GFandMySQL_Par

glassfish and mysql part

CommunityOne East - Agenda | GlassFish and MySQL, Part 2: ... | GlassFish and MySQL, Part...

Sun Java Solaris Communities My SDN Account Join SDN

Sun Developer Network (SDN)

APIs Downloads Products Support Training Participate

» search tips Search

SDN Home > Java Technology > Reference > Technical Articles and Tips >

Article

GlassFish and MySQL, Part 4: Creating a RESTful Web Service and JavaFX Client

Print-friendly Version

By Ed Ort and Carol McDonald, March 2009

Articles Index

Introduction | The Application | Building the Application

This is the fourth article in a series of articles on GlassFish and MySQL. Part 1 of the series describes the advantages of using GlassFish with MySQL and illustrates why the combination is a perfect choice for developing and deploying web applications.

Part 2 shows how to develop a create, read, update, and delete (CRUD) web application that uses GlassFish and MySQL. The application uses the Java Persistence API implemented in GlassFish to manage data persistence.

Part 3 shows how easy it is to convert the controller layer of the web application, that is, the layer of the application that performs the CRUD operations -- into a SOAP-based web service. It also shows how to create a client for the web service.

In Part 4, you'll learn how to create a RESTful web service for the web application. You'll also examine a JavaFX client for the RESTful web service. As was the case for Part 3, the web service discussed in Part 4 uses GlassFish, MySQL, and the Java Persistence API.

This article shows you how to use the NetBeans IDE with GlassFish and MySQL to create the RESTful web service. Specifically, you'll take advantage of features in NetBeans IDE 6.5, GlassFish Server v2.1, and MySQL 5.1 Community Server to build and deploy the RESTful web service. This article also shows you how to use the NetBeans IDE with JavaFX support to run a JavaFX client for the web service.

This article shows you how to use the NetBeans IDE with GlassFish and MySQL to create a RESTful web

Get Training

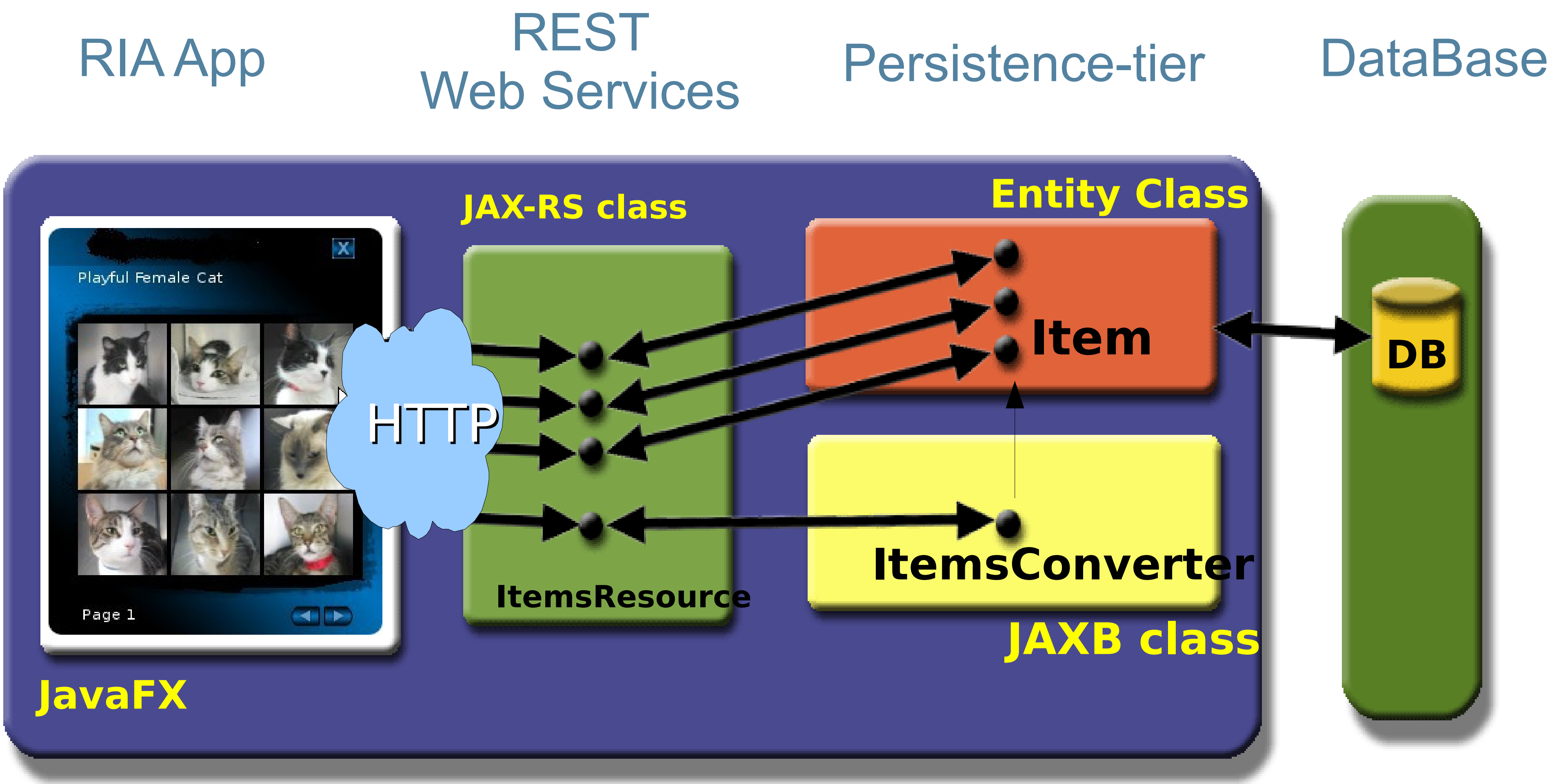
- Sun Certified Developer for Java Web Services Learning Path
- MySQL Developer Learning Path
- Getting Started with the Java Persistence API

Ed Ort is a writer on the staff of the Sun Developer Network. He has written extensively about a wide variety of programming topics including relational database technology, programming languages, web services, and Ajax. Read his blog.

Carol McDonald is a Java Technology evangelist at Sun. Her prolific blogging and

Done

RESTful Catalog



In Conclusion

Understand the storage engines

Keep data types small

Data size = Disk I/O = Memory = Performance

Know your SQL

Use EXPLAIN , use the Query Analyzer

Understand the query optimizer

Use good indexing

Resources

MySQL Forge and the Forge Wiki

http://forge.mysql.com/

Planet MySQL

http://planetmysql.org/

MySQL DevZone

http://dev.mysql.com/

High Performance MySQL book

http://java.sun.com/developer/technicalArticles/gla

http://java.sun.com/developer/technicalArticles/gla