CSE544: SQL

Monday 3/27 and Wednesday 3/29, 2006

SQL Introduction

Standard language for querying and manipulating data

Structured **Q**uery **L**anguage

Many standards out there:

- ANSI SQL, SQL92 (a.k.a. SQL2), SQL99 (a.k.a. SQL3), ….
- Vendors support various subsets: watch for fun discussions in class !

SQL

- Data Definition Language (DDL)
  - Create/alter/delete tables and their attributes
  - Following lectures...
- Data Manipulation Language (DML)
  - Query one or more tables – discussed next !
  - Insert/delete/modify tuples in tables

Tables in SQL

Table name

Attribute names

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Tuples or rows

Tables Explained

- The *schema* of a table is the table name and its attributes:

Product(PName, Price, Category, Manfacturer)

- A *key* is an attribute whose values are unique; we underline a key

Product(<u>PName</u>, Price, Category, Manfacturer)

Data Types in SQL

- Atomic types:
  - Characters: CHAR(20), VARCHAR(50)
  - Numbers: INT, BIGINT, SMALLINT, FLOAT
  - Others: MONEY, DATETIME, …
- Every attribute must have an atomic type
  - Hence tables are flat
  - Why ?

Tables Explained

- A tuple = a record
  - Restriction: all attributes are of atomic type
- A table = a set of tuples
  - Like a list…
  - …but it is unorderd:
    no **first()**, no **next()**, no **last()**.

SQL Query

Basic form: (plus many many more bells and whistles)

```
SELECT  <attributes>
FROM    <one or more relations>
WHERE   <conditions>
```

Simple SQL Query

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT   *
FROM     Product
WHERE    category='Gadgets'
```

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks

“selection”

Simple SQL Query

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT   PName, Price, Manufacturer
FROM     Product
WHERE    Price > 100
```

PName	Price	Manufacturer
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

"selection" and "projection"

Notation

Input Schema

Product(<u>PName</u>, Price, Category, Manfacturer)

```
SELECT   PName, Price, Manufacturer
FROM     Product
WHERE    Price > 100
```

Answer(PName, Price, Manfacturer)

Output Schema

Details

- Case insensitive:
  - Same: SELECT Select select
  - Same: Product   product
  - Different: ‘Seattle’ ‘seattle’

- Constants:
  - ‘abc’ - yes
  - “abc” - no

The **LIKE** operator

```
SELECT   *
FROM     Products
WHERE    PName LIKE '%gizmo%'
```

- s **LIKE** p: pattern matching on strings
- p may contain two special symbols:
  - % = any sequence of characters
  - _ = any single character

Eliminating Duplicates

```
SELECT   DISTINCT category
FROM     Product
```

Category
Gadgets
Photography
Household

Compare to:

```
SELECT   category
FROM     Product
```

Category
Gadgets
Gadgets
Photography
Household

Ordering the Results

```
SELECT   pname, price, manufacturer
FROM     Product
WHERE    category='gizmo' AND price > 50
ORDER BY price, pname
```

Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT   DISTINCT category
FROM     Product
ORDER BY category
```

?

```
SELECT   Category
FROM     Product
ORDER BY PName
```

?

```
SELECT   DISTINCT category
FROM     Product
ORDER BY PName
```

?

Keys and Foreign Keys

Company

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

Key

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Foreign key

Joins

Product (<u>pname</u>, price, category, manufacturer)
Company (<u>cname</u>, stockPrice, country)

Find all products under $200 manufactured in Japan;
return their names and prices.

Join
between Product
and Company

```
SELECT   PName, Price
FROM     Product, Company
WHERE    Manufacturer=CName AND Country='Japan'
         AND Price <= 200
```

Joins

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Company

Cname	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT   PName, Price
FROM     Product, Company
WHERE    Manufacturer=CName AND Country='Japan'
         AND Price <= 200
```

PName	Price
SingleTouch	$149.99

More Joins

Product (pname, price, category, manufacturer)
Company (cname, stockPrice, country)

Find all Chinese companies that manufacture products both in the 'electronic' and 'toy' categories

```
SELECT   cname

FROM

WHERE
```

A Subtlety about Joins

Product (<u>pname</u>, price, category, manufacturer)
Company (<u>cname</u>, stockPrice, country)

Find all countries that manufacture some product in the ‘Gadgets’ category.

```
SELECT   Country
FROM     Product, Company
WHERE    Manufacturer=CName AND Category=‘Gadgets’
```

Unexpected duplicates

A Subtlety about Joins

Product

Name	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Company

Cname	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT   Country
FROM     Product, Company
WHERE    Manufacturer=CName AND Category='Gadgets'
```

Country
??
??

What is the problem ? What's the solution ?

Tuple Variables

Person(<u>pname</u>, address, worksfor)
Company(<u>cname</u>, address)

```
SELECT   DISTINCT pname, address
FROM     Person, Company
WHERE    worksfor = cname
```

Which address ?

```
SELECT   DISTINCT Person.pname, Company.address
FROM     Person, Company
WHERE    Person.worksfor = Company.cname
```

```
SELECT   DISTINCT x.pname, y.address
FROM     Person AS x, Company AS y
WHERE    x.worksfor = y.cname
```

Meaning (Semantics) of SQL Queries

```
SELECT a1, a2, …, ak
FROM    R1 AS x1, R2 AS x2, …, Rn AS xn
WHERE  Conditions
```

Answer = {}
for x_1 **in** R_1 **do**
    for x_2 **in** R_2 **do**
        …..
            for x_n **in** R_n **do**
                if Conditions
                    then Answer = Answer $\cup$ $\{(a_1,\ldots,a_k)\}$
return Answer

An Unintuitive Query

```
SELECT  DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A   OR   R.A=T.A
```

What does it compute ?

Computes $R \cap (S \cup T)$

But what if $S = \phi$?

Subqueries Returning Relations

Company(<u>name</u>, city)

Product(<u>pname</u>, maker)

Purchase(<u>id</u>, product, buyer)

Return cities where one can find companies that manufacture products bought by Joe Blow

```
SELECT  Company.city
FROM    Company
WHERE  Company.name  IN
                (SELECT Product.maker
                 FROM   Purchase , Product
                 WHERE Product.pname=Purchase.product
                       AND Purchase .buyer = ‘Joe Blow‘);
```

Subqueries Returning Relations

Is it equivalent to this ?

```
SELECT  Company.city
FROM      Company, Product, Purchase
WHERE   Company.name= Product.maker
          AND  Product.pname  = Purchase.product
          AND  Purchase.buyer = ‘Joe Blow’
```

Beware of duplicates !

Removing Duplicates

```
SELECT DISTINCT Company.city
FROM    Company
WHERE  Company.name  IN
                (SELECT Product.maker
                  FROM   Purchase , Product
                  WHERE Product.pname=Purchase.product
                       AND Purchase .buyer = ‘Joe Blow‘);
```

```
SELECT DISTINCT Company.city
FROM     Company, Product, Purchase
WHERE   Company.name= Product.maker
          AND  Product.pname  = Purchase.product
          AND  Purchase.buyer = ‘Joe Blow’
```

Now they are equivalent

Subqueries Returning Relations

You can also use: s > ALL R
s > ANY R
EXISTS R

Product (pname, price, category, maker)

Find products that are more expensive than all those produced By "Gizmo-Works"

```
SELECT  name
FROM    Product
WHERE   price >  ALL (SELECT price
                      FROM   Purchase
                      WHERE  maker='Gizmo-Works')
```

Question for Database Fans and their Friends

- Can we express this query as a single SELECT-FROM-WHERE query, without subqueries ?

Question for Database Fans and their Friends

- Answer: all SFW queries are monotone (figure out what this means). A query with **ALL** is not monotone

Correlated Queries

Movie (title, year, director, length)

Find movies whose title appears more than once.

correlation

```
SELECT DISTINCT title
FROM   Movie AS x
WHERE  year <> ANY
                    (SELECT  year
                     FROM    Movie
                     WHERE   title =  x.title);
```

Note (1) scope of variables (2) this can still be expressed as single SFW

Complex Correlated Query

Product (pname, price, category, maker, year)

- Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

```
SELECT DISTINCT  pname, maker
FROM      Product AS x
WHERE  price > ALL  (SELECT  price
                                   FROM    Product AS y
                                   WHERE  x.maker = y.maker AND y.year < 1972);
```

Very powerful ! Also much harder to optimize.

Aggregation

```
SELECT  avg(price)
FROM    Product
WHERE   maker="Toyota"
```

```
SELECT  count(*)
FROM    Product
WHERE   year > 1995
```

SQL supports several aggregation operations:

sum, count, min, max, avg

Except count, all aggregations apply to a single attribute

Aggregation: Count

COUNT applies to duplicates, unless otherwise stated:

```
SELECT  Count(category)
FROM    Product
WHERE   year > 1995
```

same as Count(*)

We probably want:

```
SELECT  Count(DISTINCT category)
FROM    Product
WHERE   year > 1995
```

More Examples

Purchase(product, date, price, quantity)

```
SELECT  Sum(price * quantity)
FROM    Purchase
```

What do they mean ?

```
SELECT  Sum(price * quantity)
FROM    Purchase
WHERE   product = ‘bagel’
```

Simple Aggregations

Purchase

Product	Date	Price	Quantity
Bagel	10/21	1	20
Banana	10/3	0.5	10
Banana	10/10	1	10
Bagel	10/25	1.50	20

```
SELECT  Sum(price * quantity)
FROM    Purchase
WHERE   product = ‘bagel’
```

⇒ 50 (= 20+30)

Grouping and Aggregation

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product.

```
SELECT      product, Sum(price*quantity) AS TotalSales
FROM        Purchase
WHERE       date > '10/1/2005'
GROUP BY    product
```

Let's see what this means…

Grouping and Aggregation

1. Compute the FROM and WHERE clauses.

2. Group by the attributes in the GROUPBY

3. Compute the SELECT clause: grouped attributes and aggregates.

1&2. FROM-WHERE-GROUPBY

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

3. SELECT

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

Product	TotalSales
Bagel	50
Banana	15

```
SELECT      product, Sum(price*quantity) AS TotalSales
FROM        Purchase
WHERE       date > '10/1/2005'
GROUP BY    product
```

GROUP BY v.s. Nested Quereis

```
SELECT      product, Sum(price*quantity) AS TotalSales
FROM        Purchase
WHERE       date > ‘10/1/2005’
GROUP BY    product
```

```
SELECT DISTINCT  x.product, (SELECT Sum(y.price*y.quantity)
                             FROM     Purchase y
                             WHERE x.product = y.product
                                   AND y.date > ‘10/1/2005’)
                            AS TotalSales
FROM        Purchase x
WHERE       x.date > ‘10/1/2005’
```

Another Example

What does it mean ?

```
SELECT     product,
           sum(price * quantity) AS SumSales
           max(quantity) AS MaxQuantity
FROM       Purchase
GROUP BY   product
```

HAVING Clause

Same query, except that we consider only products that had at least 100 buyers.

```
SELECT      product, Sum(price * quantity)
FROM        Purchase
WHERE       date > '10/1/2005'
GROUP BY    product
HAVING      Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.

General form of Grouping and Aggregation

SELECT S

FROM $R_1,\ldots,R_n$

WHERE C1

GROUP BY $a_1,\ldots,a_k$

HAVING C2

Why ?

S = may contain attributes $a_1,\ldots,a_k$ and/or any aggregates but NO OTHER ATTRIBUTES

C1 = is any condition on the attributes in $R_1,\ldots,R_n$

C2 = is any condition on aggregate expressions

General form of Grouping and Aggregation

```
SELECT    S
FROM      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

Evaluation steps:

1. Evaluate FROM-WHERE, apply condition C1
2. Group by the attributes $a_1,\ldots,a_k$
3. Apply condition C2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

Advanced SQLizing

1. Getting around INTERSECT and EXCEPT

2. Quantifiers

3. Aggregation v.s. subqueries

INTERSECT and EXCEPT: not in SQL Server

1. INTERSECT and EXCEPT:

If R, S have no duplicates, then can write without subqueries (HOW ?)

```
(SELECT R.A, R.B
FROM    R)
   INTERSECT
(SELECT S.A, S.B
FROM    S)
```

```
SELECT R.A, R.B
FROM    R
WHERE
    EXISTS(SELECT *
               FROM S
               WHERE R.A=S.A and R.B=S.B)
```

```
(SELECT R.A, R.B
FROM    R)
   EXCEPT
(SELECT S.A, S.B
FROM    S)
```

```
SELECT R.A, R.B
FROM    R
WHERE
   NOT  EXISTS(SELECT *
               FROM S
               WHERE R.A=S.A and R.B=S.B)
```

2. Quantifiers

Product (pname, price, company)
Company(cname, city)

Find all companies that make <u>some</u> products with price < 100

```
SELECT DISTINCT  Company.cname
FROM     Company, Product
WHERE  Company.cname = Product.company and Product.price < 100
```

Existential: easy ! ☺

2. Quantifiers

Product (pname, price, company)
Company(cname, city)

Find all companies that make <u>only</u> products with price < 100

same as:

Find all companies s.t. <u>all</u> of their products have price < 100

Universal: hard ! ☹

2. Quantifiers

1. Find *the other* companies: i.e. s.t. <u>some</u> product $\geq$ 100

```
SELECT DISTINCT  Company.cname
FROM     Company
WHERE  Company.cname IN (SELECT Product.company
                          FROM Product
                          WHERE Produc.price >= 100
```

2. Find all companies s.t. <u>all</u> their products have price < 100

```
SELECT DISTINCT  Company.cname
FROM     Company
WHERE  Company.cname NOT IN (SELECT Product.company
                              FROM Product
                              WHERE Produc.price >= 100
```

3. Group-by v.s. Nested Query

Author(<u>login</u>,name)

Wrote(login,url)

- Find authors who wrote ≥ 10 documents:
- Attempt 1: with nested queries

This is SQL by a novice

```
SELECT DISTINCT Author.name
FROM        Author
WHERE       count(SELECT Wrote.url
                  FROM Wrote
                  WHERE Author.login=Wrote.login)
            > 10
```

3. Group-by v.s. Nested Query

- Find all authors who wrote at least 10 documents:
- Attempt 2: SQL style (with GROUP BY)

```
SELECT     Author.name
FROM       Author, Wrote
WHERE      Author.login=Wrote.login
GROUP BY   Author.name
HAVING     count(wrote.url) > 10
```

This is SQL by an expert

No need for DISTINCT: automatically from GROUP BY

3. Group-by v.s. Nested Query

Author(login,name)

Wrote(login,url)

Mentions(url,word)

Find authors with vocabulary ≥ 10000 words:

```
SELECT     Author.name
FROM       Author, Wrote, Mentions
WHERE      Author.login=Wrote.login AND Wrote.url=Mentions.url
GROUP BY   Author.name
HAVING     count(distinct Mentions.word) > 10000
```

Two Examples

Store(sid, sname)
Product(pid, pname, price, sid)

Find all stores that sell *only* products with price > 100

same as:

Find all stores s.t. all their products have price > 100)

```
SELECT Store.name
FROM    Store, Product
WHERE  Store.sid = Product.sid
GROUP BY  Store.sid, Store.name
HAVING 100 < min(Product.price)
```

Why both ?

Almost equivalent…

```
SELECT Store.name
FROM    Store
WHERE
   100 < ALL (SELECT Product.price
                      FROM product
                      WHERE Store.sid = Product.sid)
```

```
SELECT Store.name
FROM    Store
WHERE  Store.sid NOT IN
             (SELECT Product.sid
               FROM Product
               WHERE  Product.price <= 100)
```

Two Examples

Store(<u>sid</u>, sname)
Product(<u>pid</u>, pname, price, sid)

For each store,
find its most expensive product

Two Examples

This is easy but doesn't do what we want:

```
SELECT Store.sname, max(Product.price)
FROM   Store, Product
WHERE  Store.sid = Product.sid
GROUP BY  Store.sid, Store.sname
```

Better:

```
SELECT Store.sname, x.pname
FROM   Store, Product x
WHERE  Store.sid = x.sid and
       x.price >=
           ALL (SELECT y.price
                FROM Product y
                WHERE Store.sid = y.sid)
```

But may return multiple product names per store

Two Examples

Finally, choose some pid arbitrarily, if there are many with highest price:

```
SELECT Store.sname, max(x.pname)
FROM    Store, Product x
WHERE   Store.sid = x.sid and
        x.price >=
              ALL (SELECT y.price
                   FROM Product y
                   WHERE Store.sid = y.sid)
GROUP BY Store.sname
```

NULLS in SQL

- Whenever we don’t have a value, we can put a NULL
- Can mean many things:
  - Value does not exists
  - Value exists but is unknown
  - Value not applicable
  - Etc.
- The schema specifies for each attribute if can be null (*nullable* attribute) or not
- How does SQL cope with tables that have NULLs ?

Null Values

- If x= NULL then 4*(3-x)/7 is still NULL

- If x= NULL then x=“Joe” is UNKNOWN
- In SQL there are three boolean values:

FALSE	=	0
UNKNOWN	=	0.5
TRUE	=	1

Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

```
SELECT *
FROM Person
WHERE  (age < 25) AND
       (height > 6 OR weight > 190)
```

E.g.
age=20
heigth=NULL
weight=200

Null Values

Unexpected behavior:

```
SELECT *
FROM    Person
WHERE  age < 25  OR  age >= 25
```

Some Persons are not included !

Null Values

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

```
SELECT *
FROM    Person
WHERE  age < 25  OR  age >= 25 OR age IS NULL
```

Now it includes all Persons

Outerjoins

Explicit joins in SQL = “inner joins”:

Product(name, category)
Purchase(prodName, store)

```
SELECT Product.name, Purchase.store
FROM   Product JOIN Purchase ON
                   Product.name = Purchase.prodName
```

Same as:

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

But Products that never sold will be lost !

Outerjoins

Left outer joins in SQL:

Product(name, category)

Purchase(prodName, store)

```
SELECT Product.name, Purchase.store
FROM    Product LEFT OUTER JOIN Purchase ON
                Product.name = Purchase.prodName
```

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Name	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

Application

Compute, for each product, the total number of sales in ‘September’

Product(<u>name</u>, category)

Purchase(prodName, month, store)

```
SELECT Product.name, count(*)
FROM     Product, Purchase
WHERE   Product.name = Purchase.prodName
        and  Purchase.month = ‘September’
GROUP BY Product.name
```

What’s wrong ?

Application

Compute, for each product, the total number of sales in 'September'

Product(name, category)

Purchase(prodName, month, store)

```
SELECT Product.name, count(*)
FROM     Product LEFT OUTER JOIN Purchase ON
                    Product.name = Purchase.prodName
              and  Purchase.month = 'September'
GROUP BY Product.name
```

Now we also get the products who sold in 0 quantity

Outer Joins

- Left outer join:
  - Include the left tuple even if there’s no match
- Right outer join:
  - Include the right tuple even if there’s no match
- Full outer join:
  - Include the both left and right tuples even if there’s no match

Modifying the Database

Three kinds of modifications

- Insertions
- Deletions
- Updates

Sometimes they are all called “updates”

Insertions

General form:

```
INSERT INTO R(A1,…., An) VALUES (v1,…., vn)
```

Example: Insert a new purchase to the database:

```
INSERT INTO Purchase(buyer, seller, product, store)
       VALUES ('Joe', 'Fred', 'wakeup-clock-espresso-machine',
               'The Sharper Image')
```

Missing attribute → NULL.
May drop attribute names if give them in order.

Insertions

```
INSERT  INTO  PRODUCT(name)

    SELECT DISTINCT Purchase.product
    FROM     Purchase
    WHERE   Purchase.date > “10/26/01”
```

The query replaces the VALUES keyword.
Here we insert *many* tuples into PRODUCT

Insertion: an Example

Product(name, listPrice, category)
Purchase(prodName, buyerName, price)

prodName is foreign key in Product.name

Suppose database got corrupted and we need to fix it:

Product

name	listPrice	category
gizmo	100	gadgets

Purchase

prodName	buyerName	price
camera	John	200
gizmo	Smith	80
camera	Smith	225

Task: insert in Product all prodNames from Purchase

Insertion: an Example

```
INSERT   INTO   Product(name)

 SELECT  DISTINCT  prodName
 FROM     Purchase
 WHERE   prodName  NOT IN (SELECT  name FROM  Product)
```

name	listPrice	category
gizmo	100	Gadgets
camera	-	-

Insertion: an Example

```
INSERT INTO Product(name, listPrice)

SELECT DISTINCT prodName, price
FROM Purchase
WHERE prodName NOT IN (SELECT name FROM Product)
```

name	listPrice	category
gizmo	100	Gadgets
camera	200	-
camera ??	225 ??	-

← Depends on the implementation

Deletions

Example:

```
DELETE   FROM   PURCHASE

WHERE    seller = 'Joe'   AND
         product = 'Brooklyn Bridge'
```

Factoid about SQL: there is no way to delete only a single occurrence of a tuple that appears twice in a relation.

Updates

Example:

```
UPDATE   PRODUCT
SET    price = price/2
WHERE  Product.name  IN
                 (SELECT product
                  FROM    Purchase
                  WHERE  Date ='Oct, 25, 1999');
```

An introduction to MongoDB

Sathish Ravikumar

SCHOLAR IT SOLUTIONS

mongoDB®

Agenda

1. SQL vs NoSQL
2. Introduction of MongoDB
3. MongoDB Features
4. Replication/ High Availability
5. Sharding/ Scaling

www.scholaritinc.com

Pleasanton,CA,USA

For More Information Please Contact

+1 (925) 999-0678

steve@micronetgroup.com

SCHOLAR IT SOLUTIONS

SQL vs NoSQL

mongoDB®

- ✓ NoSQL (often interpreted as Not only SQL) database
- ✓ It provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases.

SQL	NoSQL
Relational Database Management System (RDBMS)	Non-relational or distributed database system.
These databases have fixed or static or predefined schema	They have dynamic schema
These databases are best suited for complex queries	These databases are not so good for complex queries
Vertically Scalable	Horizontally scalable
Follows ACID property	Follows BASE property

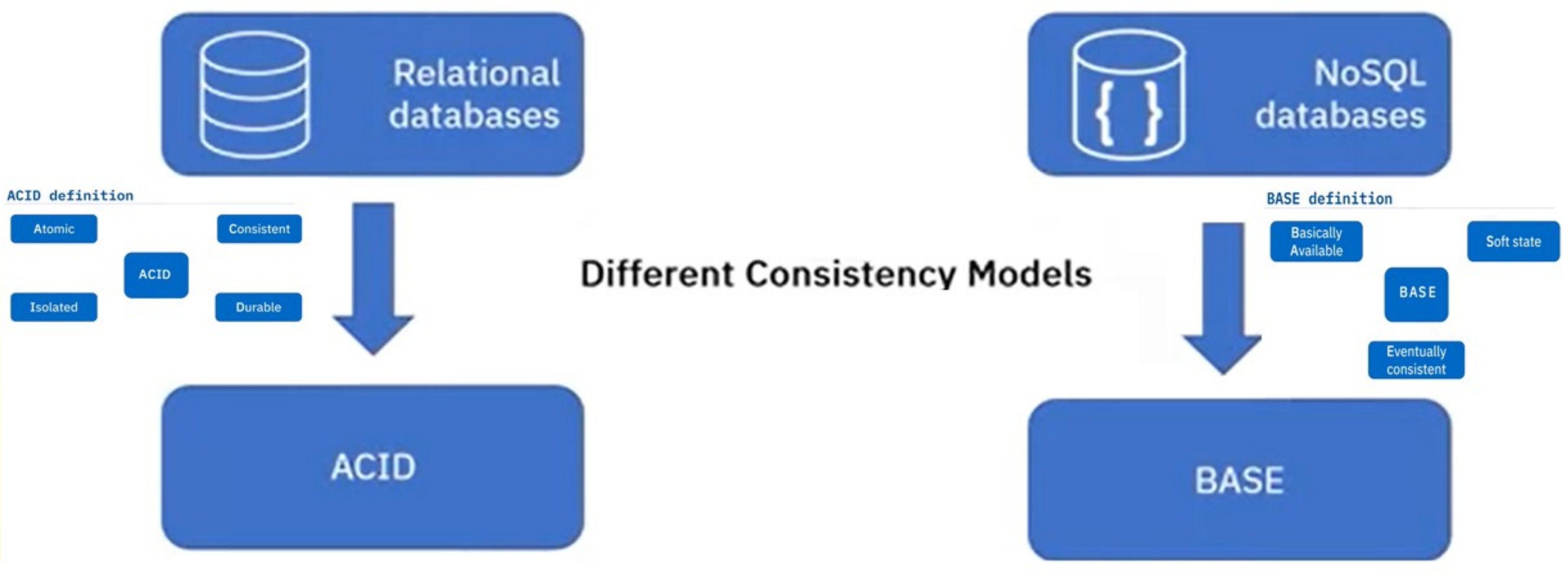
www.scholaritinc.com

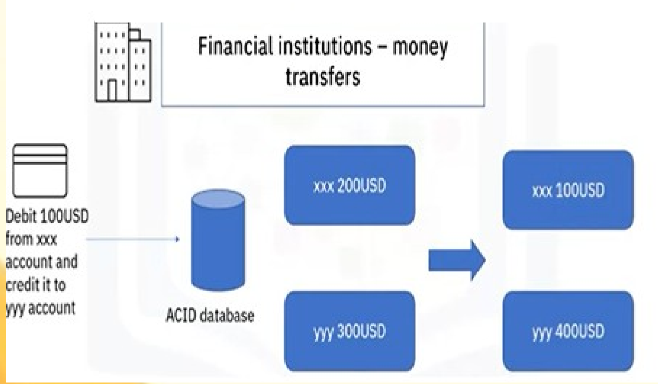
Pleasanton,CA,USA

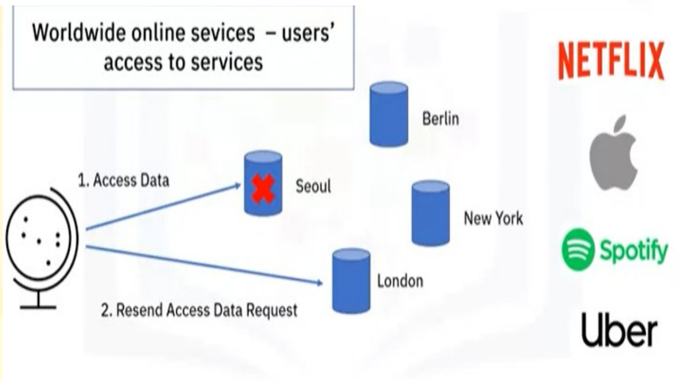
For More Information Please Contact

+1 (925) 999-0678 steve@micronetgroup.com

SQL vs NoSQL







www.scholaritinc.com
Pleasanton,CA,USA
For More Information Please Contact
+1 (925) 999-0678
steve@micronetgroup.com

SCHOLAR IT SOLUTIONS

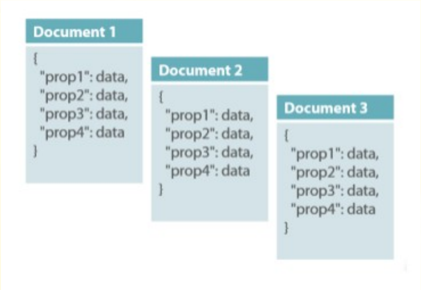
mongoDB®

NoSQL Types

Graph database

Neo4j the graph database

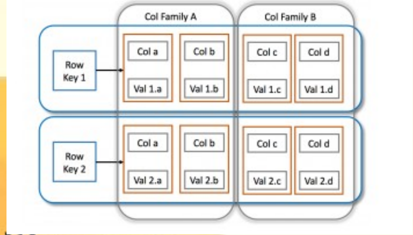
HyperGraphDB



Document-oriented

mongoDB

CouchDB relax



Column family

Cassandra

HBASE

www.scholaritinc.com

Pleasanton,CA,USA

For More Information Please Contact
+1 (925) 999-0678 steve@micronetgroup.com

mongoDB® What is MongoDB?

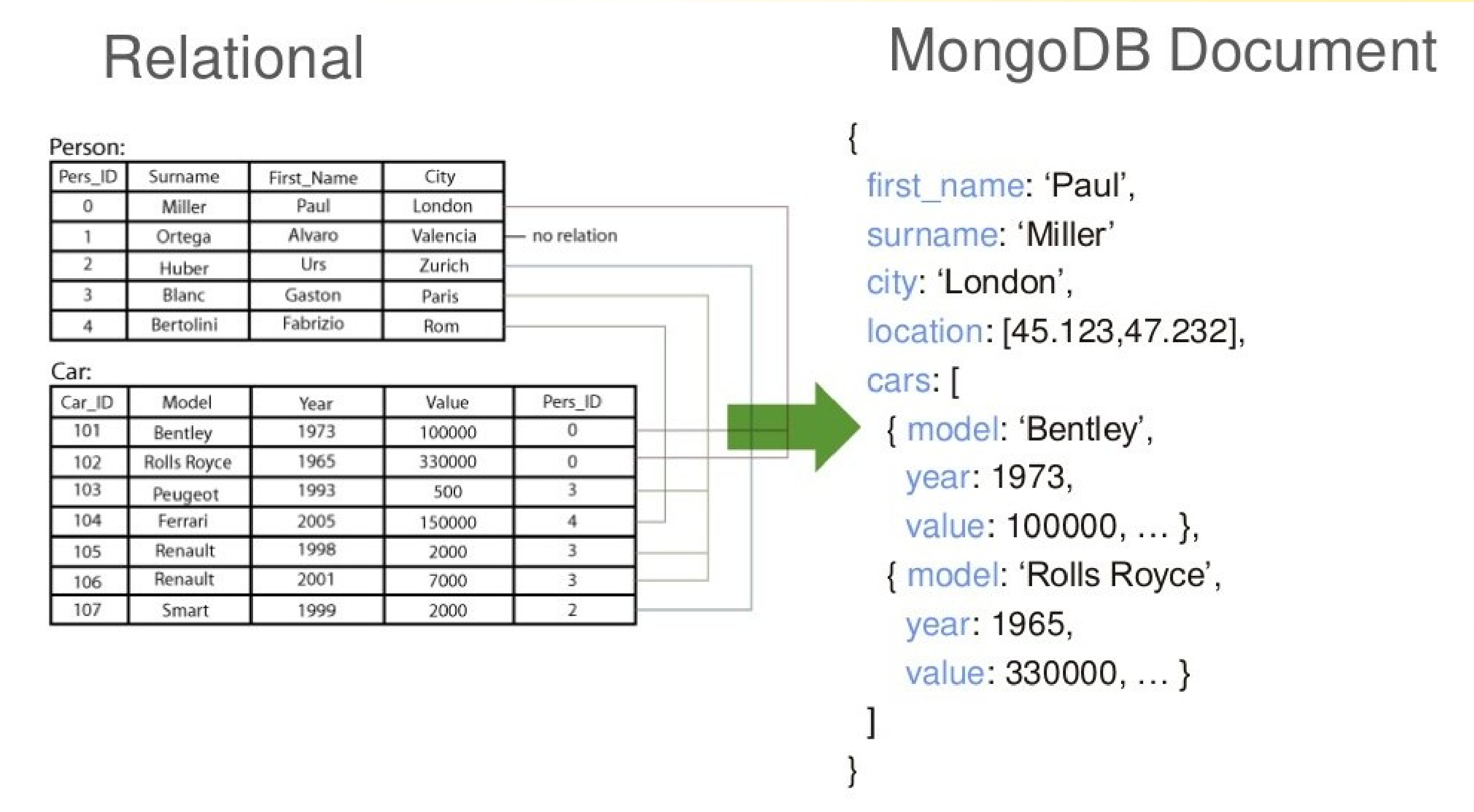
- MongoDB is an open source, document-oriented database designed with both scalability and developer agility in mind.
- Instead of storing your data in tables and rows as you would with a relational database, in MongoDB you store JSON-like documents with dynamic schemas(schema-f...

```
{
    "_id" : ObjectId("5114e0bd42…"),
    “FirstName" : "John",
    “LastName" : "Doe",
    “Age" : 39,
   “Interests" : [ "Reading",  "Mountain Biking ]
   “Favorites": {
                  "color": "Blue",
                   "sport": "Soccer“
               }
 }
```

SCHOLAR IT SOLUTIONS

mongoDB®

MongoDB is Easy to Use



Relational

Person:

Pers_ID	Surname	First_Name	City
0	Miller	Paul	London
1	Ortega	Alvaro	Valencia
2	Huber	Urs	Zurich
3	Blanc	Gaston	Paris
4	Bertolini	Fabrizio	Rom

no relation

Car:

Car_ID	Model	Year	Value	Pers_ID
101	Bentley	1973	100000	0
102	Rolls Royce	1965	330000	0
103	Peugeot	1993	500	3
104	Ferrari	2005	150000	4
105	Renault	1998	2000	3
106	Renault	2001	7000	3
107	Smart	1999	2000	2

MongoDB Document

```
{
  first_name: ‘Paul’,
  surname: ‘Miller’
  city: ‘London’,
  location: [45.123,47.232],
  cars: [
    { model: ‘Bentley’,
      year: 1973,
      value: 100000, … },
    { model: ‘Rolls Royce’,
      year: 1965,
      value: 330000, … }
  ]
}
```


www.scholaritinc.com

Pleasanton,CA,USA

For More Information Please Contact

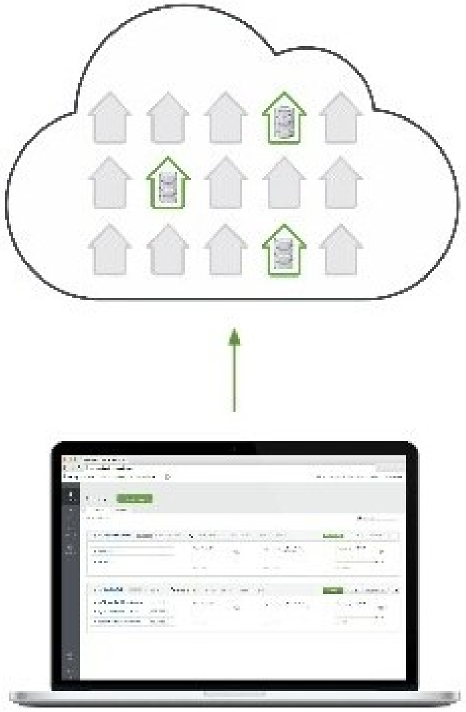
+1 (925) 999-0678 steve@micronetgroup.com

SCHOLAR IT SOLUTIONS

mongoDB®

MongoDB Atlas Benefits

Database as a service for MongoDB



Run for You	Safe & Secure	No Lock-In
• Spin up a cluster in seconds • Replicated & always-on deployments • Fully elastic: scale out or up in a few clicks with zero downtime • Automatic patches & simplified upgrades for the newest MongoDB features	• Authenticated & encrypted • Continuous backup with point-in-time recovery • Fine-grained monitoring & custom alerts	• On-demand pricing model; billed by the hour • Multi-cloud support (AWS available with others coming soon) • Part of a suite of products & services designed for all phases of your app; migrate easily to different environments (private cloud, on-prem, etc) when needed

www.scholaritinc.com

Pleasanton,CA,USA

For More Information Please Contact

+1 (925) 999-0678

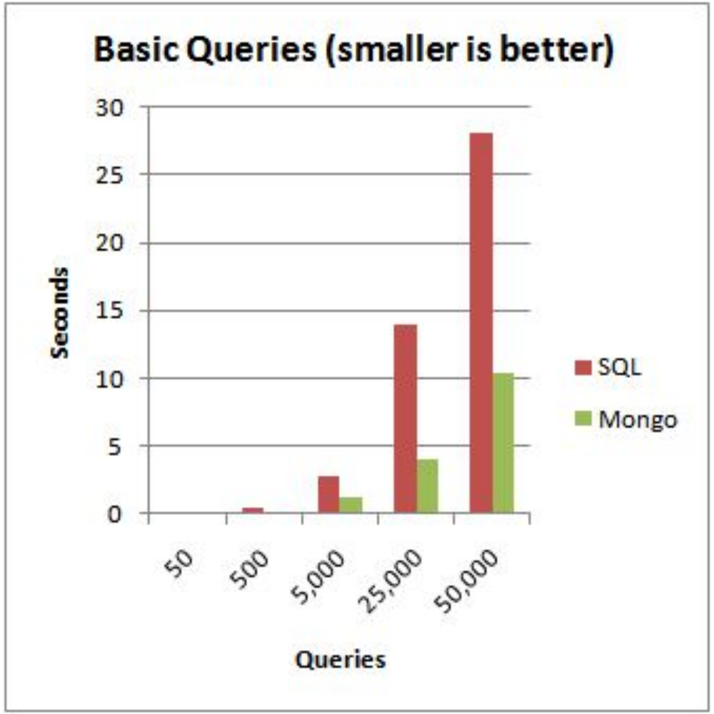
steve@micronetgroup.com

Database Popularity

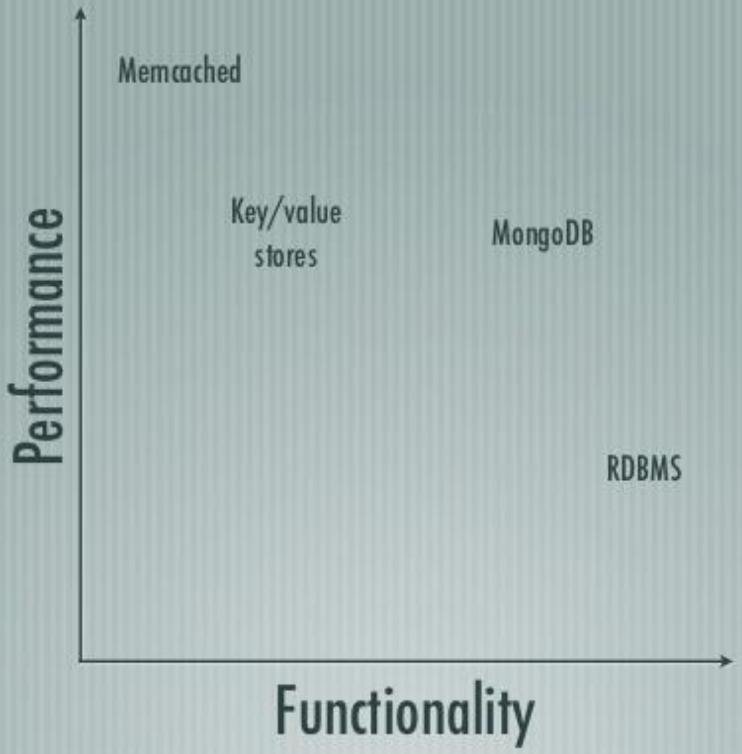
Rank	Name	Score
1.	Oracle	1617.19
2.	MySql	1254.27
3.	SQL Server	1234.46
4.	PostgreSQL	190.83
5.	DB2	165.9
6.	MongoDB	161.87
7.	Microsoft Access	141.6
8.	SQLite	78.78
9.	Sybase	77.75

http://db-engines.com/en/ranking

How Fast?



Basic Queries (smaller is better)
30
25
20
15
10
5
0
Seconds
50
500
5,000
25,000
50,000
Queries
SQL
Mongo


Memcached
Key/value
stores
MongoDB
RDBMS
Performance
Functionality

A BSON (binary JSON) document

```
{
    "_id" : ObjectId("52832eb59f36fe144eeea8dc"),
    "baseprice" : 8.99,
    "category" : "toys",
    "colors" : [ "red", "green", "cosmic purple"],
    "name" : "Cosmic Yo-yo",
    "promotions" : [
            { "coupon" : "XY678", "saleprice" : 7.99, "expires" : ISODate("2013-12-12T00:00:00Z") },
            { "coupon" : "AB8888", "saleprice" : 7.49, "expires" : ISODate("2014-01-01T00:00:00Z") }
           ]
}
```

Terminology

Database -> Database

Table -> Collection

Record / Row -> Document

Field -> Field

find()

```
db.products.findOne()
db.products.find()
db.products.find().pretty()
db.products.find({ _id : ObjectId("52832eb59f36fe144eeea8dc") })
db.products.find({ name : "Cosmic Yo-yo" })
db.products.find({ name : /^hack/i }).pretty()
```

Index fields used for find!

Projections

```
db.products.find({ "name" : "Hacky Sack Maxx" },{  baseprice : 1 } )
db.products.find({ "name": "Hacky Sack Maxx" },{  baseprice: 1, category : 1 } )
db.products.find( { }, {  promotions : 1 } )
db.products.find( { }, {  promotions : 1 } )[0].promotions
```

Queries

```
db.products.find().sort( { baseprice : -1 }).pretty()
db.products.find().limit(1).pretty()
db.products.find().limit(1).skip(1).pretty()
```

Conditions

```
db.products.find({ baseprice : { $gt : 4.99 }})
db.products.find({ baseprice : { $lte : 4.99 }})
db.products.find({ promotions : { $lte : 4.99 }})
db.products.find( { colors : { $in :  ["red"] }})
db.products.find( { "promotions.coupon" : { $in :  [  "XY678"
] }}).pretty()
db.products.find({ $and : [{ category : "toys"},{ baseprice : {
$gt : 4.99 }}]}
```

Other Queries

```
db.products.count()
db.products.find({ baseprice : { $gt : 2.99 }}).count()
db.products.insert({ name : "Juggle-O-rama", baseprice : 11.99 })
db.products.update({ name : "Juggle-O-rama" }, { $set : { category : "toys" }})
db.products.update({ name: “Juggle-O-rama” }, { $set : {  colors : ["silver", "gold"]}})
db.products.update({ name: "Juggle-O-rama" }, { $push : {  colors : "sea foam green"}})
```

Don’t forget $set!

Aggregation

db.products.aggregate({ $group : { _id : "$category", totalprice : { $sum : "$baseprice" }}})

Primary Benefits

- Speed Speed Speed!
- Rich Dynamic Queries
- Lazy Creation
- Flexible Schema
- Returns JSON
- Easy Replication and Failover
- Auto-Sharding
- MapReduce

Lazy Creation

Lazy Creation Saves Developer Time

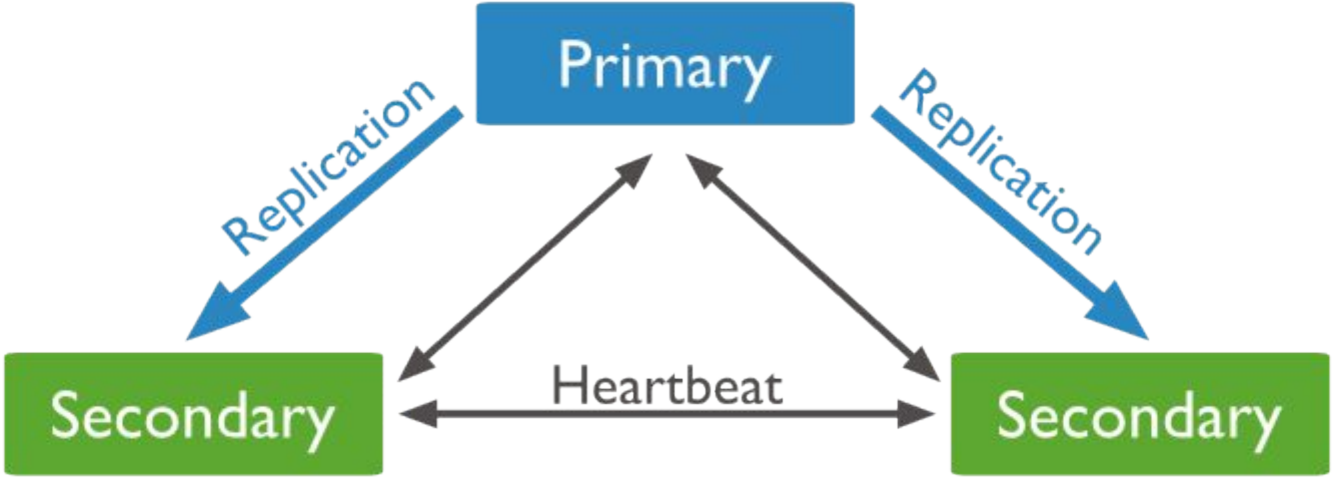
- Instant Set-up
- No Change Scripts
- Easier Data Migration
- Great for Data Warehousing

Flexible Schemas

- I.E. Different data for different for different product types
- Flexible nesting rules
- Simplifies Internationalization
- Easy Custom Fields

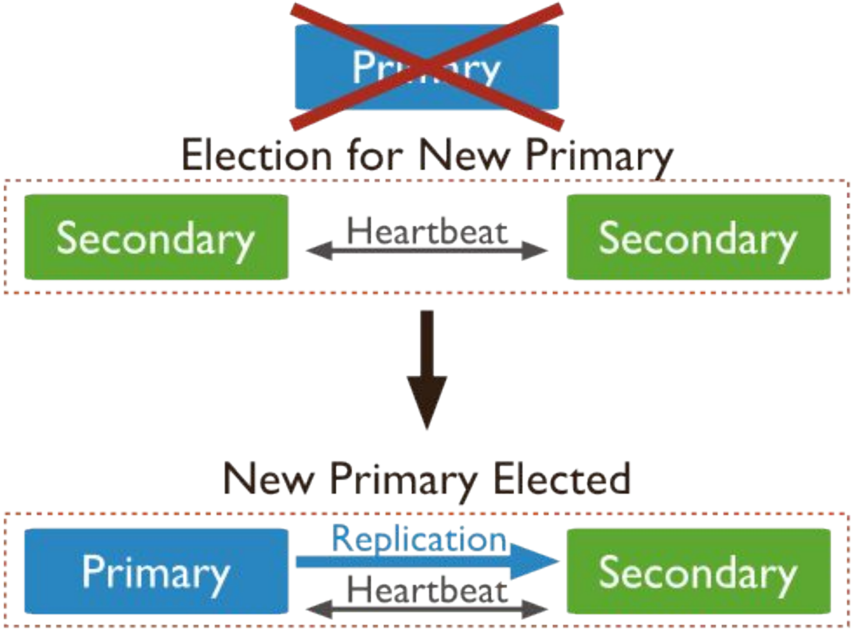
Replica Sets

Master / Slave

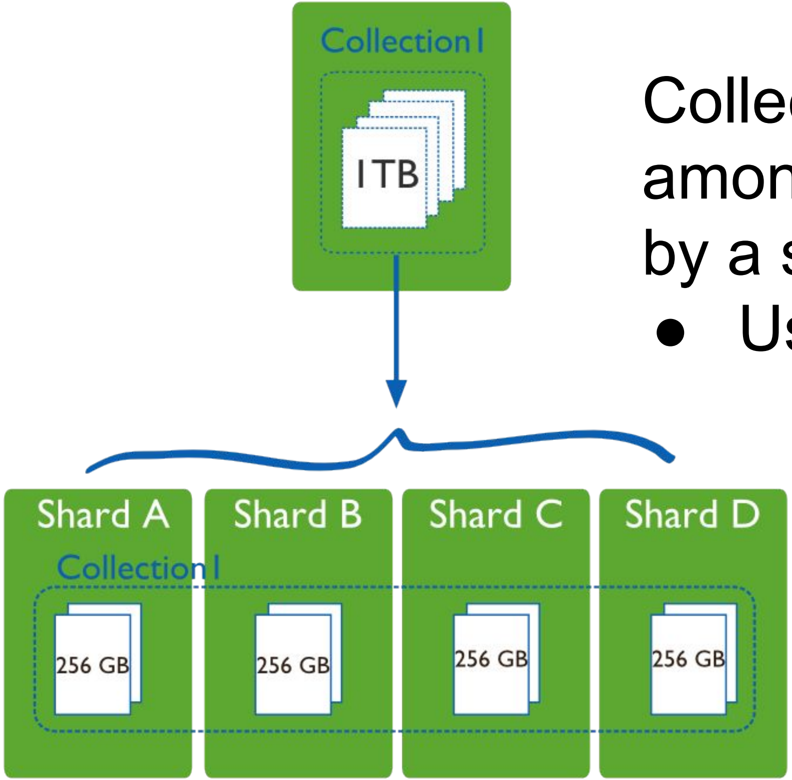


Replica Set Election

If a primary fails, the secondaries automatically elect a new primary



Sharding



Collections are split amongst multiple servers by a shard key

- Used to scale writes

Limitations

- No Transactions
- No Joins
- RAM intensive
- No referential integrity
- Eventual Consistency

Design Considerations

Embed vs. Reference

- Instead of joining junction tables, embed subdocuments in documents
- 90% of the time choose embed over reference
- You may have to store the same data twice

Denormalized Data

```
{
	_id : ObjectId(...),
	Name: “November Specials”,
	Promotions : [
			{ Title: “20% off all Yo-yos”,		Coupon: “AB345” },
			{ Title: “Free shipping on Hacky Sacks”, Coupon : “XY456” }
			],
	Dates : [ ISODate(“2013-11-01”), ISODate(“2013-11-31) ]
}, { _id : ObjectId(...),
	Name: “December Specials”,
	Promotions : [
			{ Title: “10% off all frisbees”, Coupon: “BA445” },
			{ Title: “Free overnight shipping on all jump ropes”, Coupon : “XY456” }
```

Schema Design

SQL: Optimizing how data is stored

MongoDB: Optimize how data is used

SQL: What answers do I have?

MongoDB: What questions do I have?

Choices

```
{
	OrderId(“....”)
	Items : [
			{ _id : ObjectId(“...”), name : “Cosmic Yo-yo”, color : “red”, qty : 1 },
			{ _id : ObjectId(“...”), name : Hackey Sack Maxx”, color : “tiger orange”, qty: 2
}
		]
	Promotions : [
				{ _id : ObjectId(“...”), Coupon : “AB456” : 6.99 }
			]
}
```

Use Cases

- Anything with user generated data
  - Social Media
  - CMS
  - Blogs
- Product Data (Ecommerce)
- Games
- Location services
- Logging
  - Clickstream
- Analytics
  - Real-Time
  - Data Warehouses

Not Great For

Transaction Critical Data

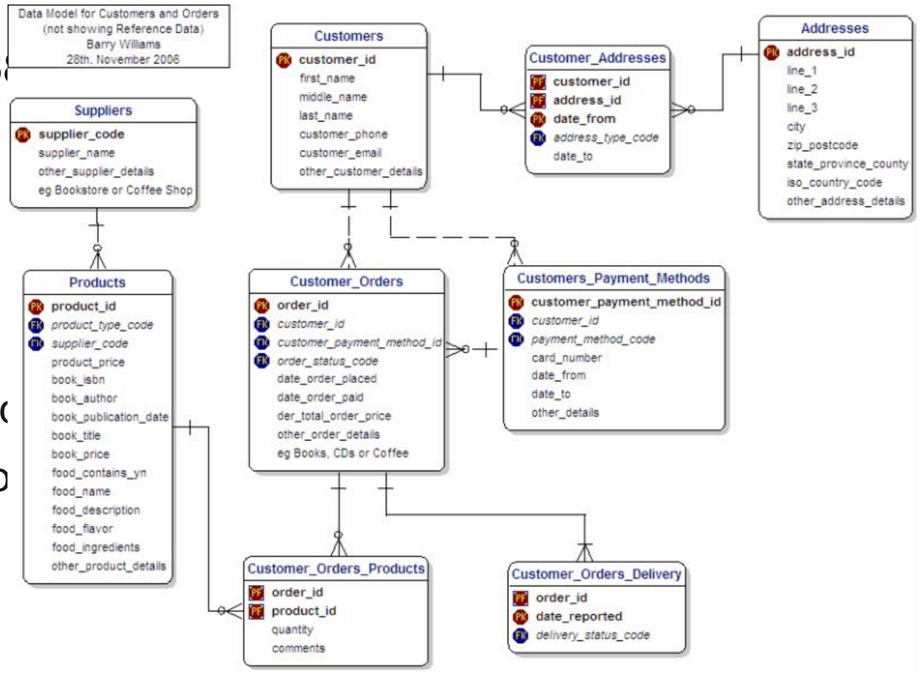
- Purchases
- Banking
- Inventory Control

Use Both!

Products in MongoDB

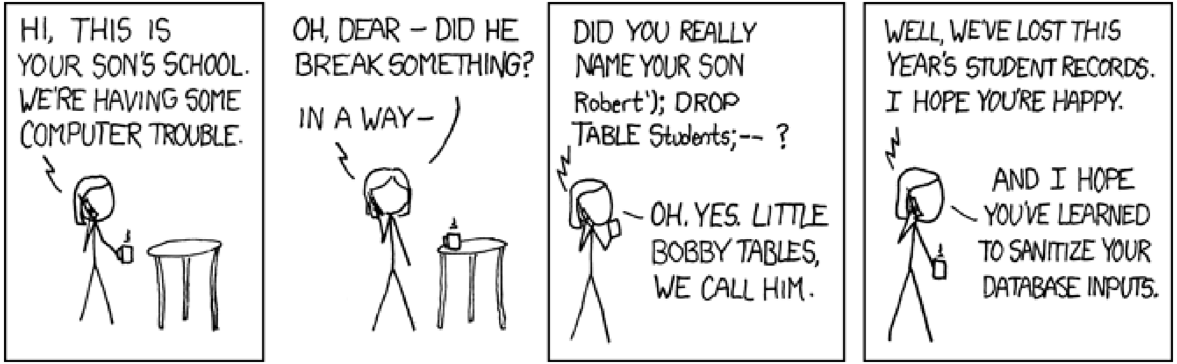
```
{
        "_id" : ObjectId("52833435add826d9da83926
        "baseprice" : 4.99,
        "category" : "toys",
        "colors" : [ "tiger orange", " canary yellow" ],
        "name" : "Hacky Sack Maxx",
        "promotions" : [ { "coupon" : "ZY678", "salepri
                                    { "coupon" : "CD8888", "promo
                        ]
}
```

Orders in SQL



Architect for Scalability

- Databases need to handle peak load, not average load
- Avoid Unnecessary Data Transformations
- Developer productivity is part of scalability



Write Python instead of SQL!

An introduction to SQLModel.

Plain DBAPI+SQL

```
>>> import psycopg2
>>> conn = psycopg2.connect('dbname=test')
>>> cur = conn.cursor()
>>> cur.execute("CREATE TABLE test (id serial PRIMARY KEY, data varchar)")
>>> cur.execute("INSERT INTO test VALUES (%s)", ("Little Bobby 'tables'",))
>>> cur.execute("INSERT INTO test VALUES ('%s')" % ("Little Bobby 'tables'",))
ProgrammingError: syntax error at or near "tables"
LINE 1: INSERT INTO test (data) VALUES ('Little Bobby 'tables'')
```

SQLModel (ORM)

```
from typing import Optional

from sqlmodel import Field, Session, SQLModel, create_engine

class Hero(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    name: str
    secret_name: str
    age: Optional[int] = None

hero_1 = Hero(name="Deadpond", secret_name="Dive Wilson")
hero_2 = Hero(name="Spider-Boy", secret_name="Pedro Parqueador")
hero_3 = Hero(name="Rusty-Man", secret_name="Tommy Sharp", age=48)


engine = create_engine("sqlite:///database.db")
SQLModel.metadata.create_all(engine)

with Session(engine) as session:
    session.add(hero_1)
    session.add(hero_2)
    session.add(hero_3)
    session.commit()
```

SQLModel (ORM)

https://sqlmodel.tiangolo.com/

What else is there

Query options to control column/relationship loading

Cascading along relationships

Custom types

Signals

Why use SQLAlchemy/ SQLModel again?

Sacrifice a little bit of performance / SQL purity for much faster development

Cleaner / shorter code

Basically no SQL injection risks

Excellent documentation

Very helpful community