



Data models and databases

Lecture-01



Introduction, Relational Algebra



Information

- ▶ Lecturer
 - Péter Lehotay-Kéry
 - lepuuai@inf.elte.hu
 - 2-507
- ▶ Tuesday 8:30-10:00
- ▶ 00-623

Textbook

- ▶ Database Systems: The Complete Book by Garcia-Molina, Jeff Ullman and Jennifer Widom
- ▶ Antoniou, G., & Van Harmelen, F. A semantic web primer
- ▶ Eric Redmond, Jim Wilson: Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement

Topics of the semester: basics

- ▶ Relational Data Model
- ▶ Core and Extended Relational Algebra
- ▶ SQL Query and Modification
- ▶ Constraints, Triggers and Views
- ▶ PSM, Oracle PL/SQL
- ▶ Entity-Relationship Model
- ▶ Design of Relational Databases
- ▶ Indexing

Topics of the semester: advanced

- ▶ Data Warehouses
- ▶ XML Databases
- ▶ XQuery
- ▶ Semantic Web, SparQL
- ▶ Graph Databases, Neo4J
- ▶ Document Store, MongoDB
- ▶ Key-Value Store, Redis
- ▶ Distributed Databases

What is a Data Model?

- ▶ 1. Mathematical representation of data
- ▶ 2. Operations on data
- ▶ 3. Constraints

Relational Data Model

- ▶ A relation is a table

Attributes (column headers)

Beer	Manufacturer
Winterbrew	Pete's
Bud Lite	Anheuser-Busch

Tuples (rows)

The diagram illustrates a relational table. At the top, the text "Attributes (column headers)" is centered above the table. Two red arrows point from this text to the first two columns of the table header. Below the header, the table contains three rows of data, each representing a tuple. The first row is "Winterbrew" and "Pete's". The second row is "Bud Lite" and "Anheuser-Busch". To the left of the table, the text "Tuples (rows)" is written vertically, with two red arrows pointing from it to the first two rows of the table body.

Types and schemas

- ▶ *Relation schema* = relation name + attributes, in order (+ types of attributes).
 - ▶ Example: Beers(name, manf) or Beers(name: string, manf: string)
- ▶ *Database* = collection of relations.
- ▶ *Database schema* = set of all relation schemas in the database.

Why relations?

- ▶ Very simple model.
- ▶ Often matches how we think about data.
- ▶ Abstract model that underlies SQL, one of the most important database languages today.

Relational model

- ▶ Logical level:
 - ▶ The relations are considered as tables.
 - ▶ The tables has unique names
 - ▶ The columns address the attributes
 - ▶ The rows represent the records
 - ▶ Rows can be interchanged, the order of rows is irrelevant
- ▶ Physical level:
 - ▶ The relations are stored in a file structure

Examples

Example 1

A	B	C
a	b	c
d	a	a
c	b	d

Example 2

B	C	A
b	c	a
a	a	d
b	d	c

In ex. 1 and ex. 2 the columns are interchanged but the same relation

Example 3

A	B	C
c	b	d
d	a	a
a	b	c

Example 4

A	B	C
c	b	d
c	b	d
a	b	c

In ex. 1 and ex. 3 the same tuples are represented in different orders but these are the same relations too.

Ex. 4 is not a relation

Defining a Database Schema

- ▶ A database schema comprises declarations for the relations (“tables”) of the database.
- ▶ Many other kinds of elements may also appear in the database schema, including views, constraints, triggers, indexes, etc.

Declaring a Relation

- ▶ Simplest form is:
- ▶ **CREATE TABLE <name> (<list of elements>);**

```
CREATE TABLE Sells (
    bar      CHAR(20) ,
    beer     VARCHAR(20) ,
    price    REAL
) ;
```

Elements of Table Declarations

- ▶ The principal element is a pair consisting of an attribute and a type.
- ▶ The most common types are:
 - ▶ INT or INTEGER (synonyms).
 - ▶ REAL or FLOAT (synonyms).
 - ▶ CHAR(n) = fixed-length string of n characters.
 - ▶ VARCHAR(n) = variable-length string of up to n characters.
 - ▶ DATE is a type, and the form of a date value is:
Example: ‘yyyy-mm-dd’ DATE ‘2002-09-30’

Example: Create Table

```
CREATE TABLE Sells (  
    bar        VARCHAR(20) ,  
    beer       VARCHAR(20) ,  
    price      REAL  
) ;
```

Other Declarations for Attributes

- ▶ Declaration for an attributes is a pair consisting of an attribute and a type.
- ▶ Other declarations we can make for an attribute are:
 1. NOT NULL means that the value for this attribute may never be NULL.
 2. DEFAULT <value> says that if there is no specific value known for this attribute's component in some tuple, use the stated <value>.

Example: Default Values

```
CREATE TABLE Drinkers (
    name VARCHAR(30) NOT NULL,
    addr VARCHAR(50) DEFAULT '3 Sesame St.',
    phone VARCHAR(16)
);
```

Effect of Defaults -- 1

- ▶ Suppose we insert the fact that Sally is a drinker, but we know neither her address nor her phone.
- ▶ An INSERT with a partial list of attributes makes the insertion possible:

```
INSERT INTO Drinkers (name)
```

```
VALUES ('Sally') ;
```

Effect of Defaults -- 2

- ▶ But what tuple appears in Drinkers?

name	addr	phone
'Sally'	'123 Sesame St'	NULL

- ▶ If we had declared phone NOT NULL, this insertion would have been rejected.

Remove a relation from schema

- ▶ Remove a relation from the database schema by:
 - ▶ `DROP TABLE <name>;`
- ▶ Example:

```
DROP TABLE Sells;
```

Query Languages: Relational Algebra

- ▶ What is an “Algebra”?
- ▶ Mathematical system consisting of:
 - ▶ *Operands* --- variables or values from which new values can be constructed.
 - ▶ *Operators* --- symbols denoting procedures that construct new values from given values.

Core Relational Algebra

- ▶ Union, intersection, and difference.
 - ▶ Usual set operations, but require both operands have the same relation schema.
- ▶ Selection: picking certain rows.
- ▶ Projection: picking certain columns.
- ▶ Products and joins: compositions of relations.
- ▶ Renaming of relations and attributes.

Union, intersection, difference

- ▶ To apply these operators the relations must have the same attributes.
- ▶ Union ($R_1 \cup R_2$): all tuples from R_1 or R_2
- ▶ Intersection ($R_1 \cap R_2$): common tuples from R_1 and R_2
- ▶ Difference ($R_1 \setminus R_2$): tuples occurring in R_1 but not in R_2

Example

Relation Sells1:

Bar	Beer	Price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50

Relation Sells2:

Bar	Beer	Price
Joe's	Bud	2.50
Jack's	Bud	2.75

Sells1 \cup Sells2:

Bar	Beer	Price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Jack's	Bud	2.75

Sells1 \cap Sells2:

Bar	Beer	Price
Joe's	Bud	2.50

Sells2 \ Sells1:

Bar	Beer	Price
Jack's	Bud	2.75

Selection

- ▶ $R_1 := \sigma_C(R_2)$
 - ▶ C is a condition (as in “if” statements) that refers to attributes of R_2 .
 - ▶ R_1 is all those tuples of R_2 that satisfy C .

Example

Relation Sells:

Bar	Beer	Price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

JoeMenu := $\sigma_{\text{bar}=\text{"Joe's"}}(\text{Sells})$:

Bar	Beer	Price
Joe's	Bud	2.50
Joe's	Miller	2.75

Projection

- ▶ $R_1 := \pi_L(R_2)$
 - ▶ L is a list of attributes from the schema of R_2 .
 - ▶ R_1 is constructed by looking at each tuple of R_2 , extracting the attributes on list L , in the order specified, and creating from those components a tuple for R_1 .
 - ▶ Eliminate duplicate tuples, if any.

Example

Relation Sells:

Bar	Beer	Price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

Prices := $\Pi_{\text{beer}, \text{price}}(\text{Sells})$:

Beer	Price
Bud	2.50
Miller	2.75
Miller	3.00

Product

- ▶ $R3 := R1 \times R2$
 - ▶ Pair each tuple $t1$ of $R1$ with each tuple $t2$ of $R2$.
 - ▶ Concatenation $t1t2$ is a tuple of $R3$.
 - ▶ Schema of $R3$ is the attributes of $R1$ and $R2$, in order.
 - ▶ But beware attribute A of the same name in $R1$ and $R2$: use $R1.A$ and $R2.A$.

Example: $R3=R1 \times R2$

► R1

A	B
1	2
3	4

► R2

B	C
5	6
7	8
9	10

$$R3=R1 \times R2$$

A	R1.B	R2.B	C
1	2	5	6
1	2	7	8
1	2	9	10
3	4	5	6
3	4	7	8
3	4	9	10

Theta-Join

- ▶ $R3 := R1 \bowtie_C R2$
 - ▶ Take the product $R1 * R2$.
 - ▶ Then apply σ_C to the result.
- ▶ As for σ , C can be any boolean-valued condition.

Example

Sells:

Bar	Beer	Price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

Bars:

Name	Address
Joe's	Maple st.
Sue's	River rd.

Barinfo = Sells \bowtie Sells.bar = Bars.name Bars

Bar	Beer	Price	Name	Address
Joe's	Bud	2.50	Joe's	Maple st.
Joe's	Miller	2.75	Joe's	Maple st.
Sue's	Bud	2.50	Sue's	River rd.
Sue's	Miller	3.00	Sue's	River rd.

Natural Join

- ▶ A frequent type of join connects two relations by:
 - ▶ Equating attributes of the same name, and
 - ▶ Projecting out one copy of each pair of equated attributes.
- ▶ Called *natural* join.
- ▶ Denoted $R_3 := R_1 \bowtie R_2$.

Example.....

Sells:

Bar	Beer	Price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

Bars:

Bar	Address
Joe's	Maple st.
Sue's	River rd.

Barinfo= Sells \bowtie Bars

Bar	Beer	Price	Address
Joe's	Bud	2.50	Maple st.
Joe's	Miller	2.75	Maple st.
Sue's	Bud	2.50	River rd.
Sue's	Miller	3.00	River rd.

Renaming

- ▶ The RENAME operator gives a new schema to a relation.
- ▶ $R1 := \rho_{1(A_1, \dots, A_n)}(R2)$ makes $R1$ be a relation with attributes A_1, \dots, A_n and the same tuples as $R2$.
- ▶ Simplified notation: $R1(A_1, \dots, A_n) := R2$.

Example

Bars:

Name	Address
Joe's	Maple st.
Sue's	River rd.

$R(\text{Bar}, \text{Address}) := \text{Bars}$

Bar	Address
Joe's	Maple st.
Sue's	River rd.

Building Complex Expressions

- ▶ Algebras allow us to express sequences of operations in a natural way
 - ▶ Example: in arithmetic --- $(x + 4)^*(y - 3)$.
- ▶ Relational algebra allows the same.
- ▶ Three notations, just as in arithmetic:
 1. Sequences of assignment statements.
 2. Expressions with several operators.
 3. Expression trees.

Sequences of Assignments

- ▶ Create temporary relation names.
- ▶ Renaming can be implied by giving relations a list of attributes.
- ▶ Example: $R3 := R1 \bowtie_C R2$ can be written:

$R4 := R1 \times R2$

$R3 := \sigma_C(R4)$

Expressions in a Single Assignment

- ▶ Example: the theta-join $R3 := R1 \bowtie_C R2$ can be written: $R3 := \sigma_C(R1 \times R2)$
- ▶ Precedence of relational operators:
 1. Unary operators --- select, project, rename --- have highest precedence, bind first.
 2. Then come products and joins.
 3. Then intersection.
 4. Finally, union and set difference bind last.
- ▶ But you can always insert parentheses to force the order you desire.

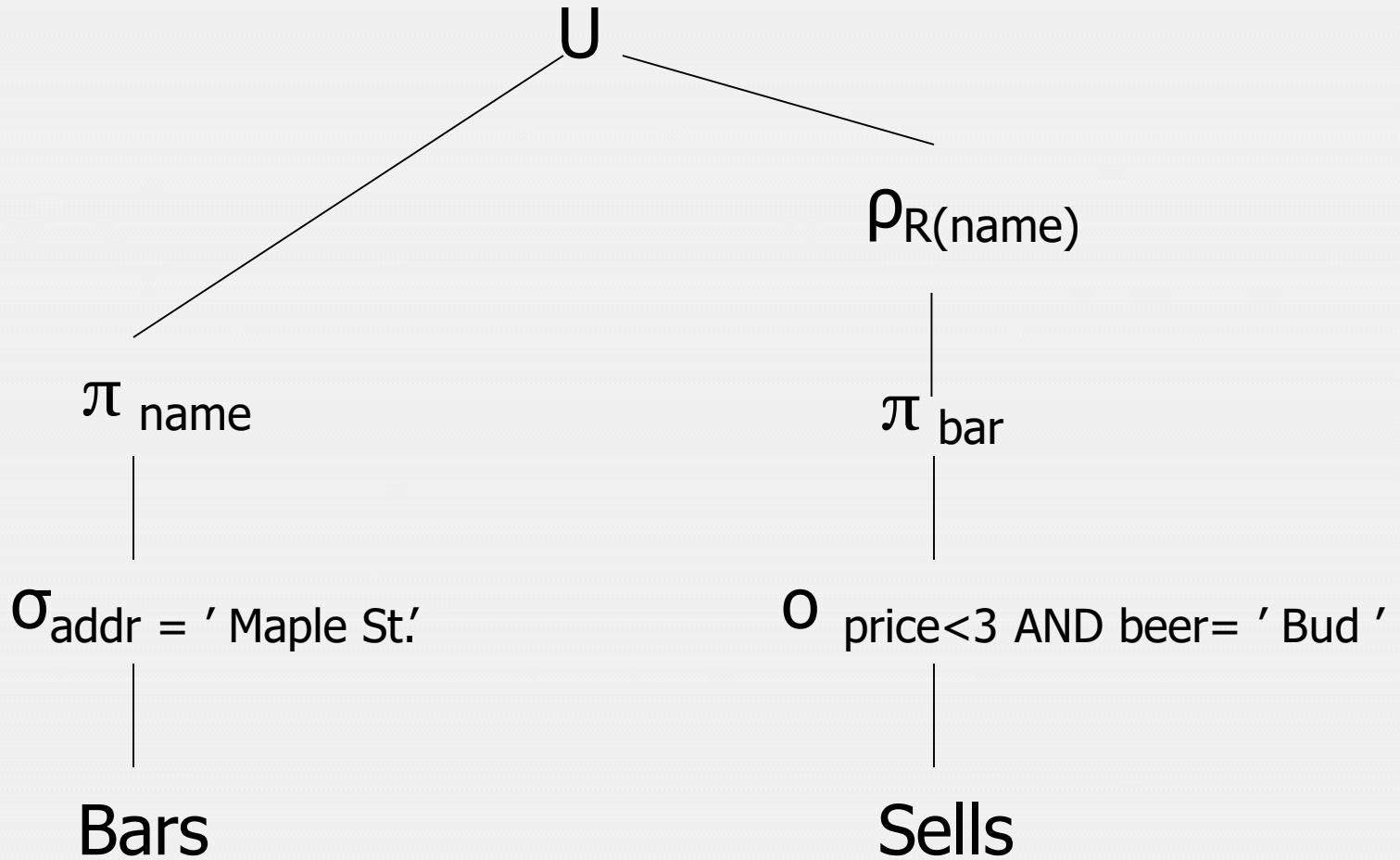
Expression Trees

- ▶ Leaves are operands --- either variables standing for relations or particular, constant relations.
- ▶ Interior nodes are operators, applied to their child or children.

Example

- ▶ Using the relations Bars(name, address) and Sells(bar, beer, price), find the names of all the bars that are either on Maple St. or sell Bud for less than \$3.

As a Tree:



Schema-Defining Rules

- ▶ For union, intersection, and difference, the schemas of the two operands must be the same, so use that schema for the result.
- ▶ Selection: schema of the result is the same as the schema of the operand.
- ▶ Projection: list of attributes tells us the schema.
- ▶ Product, Theta-join: the schema is the attributes of both relations.
 - ▶ Use R.A, etc., to distinguish two attributes named A.
- ▶ Natural join: use attributes of both relations.
 - ▶ Shared attribute names are merged.
- ▶ Renaming: the operator tells the schema.

Relational algebra: Monotonicity

- ▶ Monotone non-decreasing expression:
 - ▶ applied on more tuples, the result contains more tuples
 - ▶ Formally if $R_i \subseteq S_i$ for every $i=1,\dots,n$, then $E(R_1,\dots,R_n) \subseteq E(S_1,\dots,S_n)$.
- ▶ **Difference** is the only core expression which is not monotone:

A	B
1	0
2	1

-

A	B
1	0



A	B
1	0
2	1

-

A	B
1	0
2	1



Data models and databases

Lecture-02



Queries in SQL: SELECT

SQL

- ⌚ SQL: Structured Query Language
- ⌚ SQL is a very-high-level language, the main parts:
 - DDL data-definition language (create, drop, alter)
 - DML data-manipulation language (insert, delete, update and select). Queries: SELECT statement
- ⌚ What makes SQL viable is that its queries are “optimized” quite well, yielding efficient query executions.

Basic statement

- The principal form of a query is:

`SELECT desired attributes`

`FROM one or more tables`

`WHERE condition about tuples of the tables`

Our Running Example

- All our SQL queries will be based on the following database schema.
- *Underline indicates key attributes.*

Beers(name, manf)

Bars(name, addr, license)

Drinkers(name, addr, phone)

Likes(drinker, beer)

Sells(bar, beer, price)

Frequents(drinker, bar)

Example

- Using Beers(name, manf), what beers are made by Anheuser-Busch?

```
SELECT name  
FROM Beers  
WHERE manf = 'Anheuser-Busch' ;
```

- Relational algebraic expression:

$$\pi_{name} (\sigma_{\text{manf} = \text{'Anheuser-Busch'}} (Beers))$$

Result of Query

name
Bud
Bud Lite
Michelob

The answer is a relation with a single attribute, name, and tuples with the name of each beer by Anheuser-Busch, such as Bud.

Meaning of Single-Relation Query

- Begin with the relation in the FROM clause.
- Apply the selection indicated by the WHERE clause.
- Apply the extended projection indicated by the SELECT clause.

Operational Semantics

name	manf
Bud	Anheuser-Busch

Tuple-variable t
loops over all
tuples

Include $t.name$
in the result, if so

Check if
Anheuser-Busch

Operational Semantics --- General

- ⌚ Think of a *tuple variable* visiting each tuple of the relation mentioned in FROM.
- ⌚ Check if the “current” tuple satisfies the WHERE clause.
- ⌚ If so, compute the attributes or expressions of the SELECT clause using the components of this tuple.

(star) * in SELECT clauses

- When there is one relation in the FROM clause,
* in the SELECT clause stands for “all attributes
of this relation.”
- Example using Beers(name, manf):

```
SELECT *
FROM Beers
WHERE manf = 'Anheuser-Busch';
```

Result of Query:

name	manf
Bud	Anheuser-Busch
Bud Lite	Anheuser-Busch
Michelob	Anheuser-Busch

Now, the result has each of the attributes of Beers.

Expressions in SELECT Clauses

- Any expression that makes sense can appear as an element of a SELECT clause.
- If you want the result to have different attribute names, use “AS <new name>” to rename an attribute.
- Example: from Sells(bar, beer, price):

```
SELECT bar, beer, price * 120 AS priceInYen  
FROM Sells;
```

Result of Query

Bar	Beer	PriceinYen
Joe's	Bud	300
Joe's	Miller	330
Sue's	Bud	300
Sue's	Miller	360

Complex Conditions in WHERE Clause

- From Sells(bar, beer, price), find the price Joe's Bar charges for Bud:

```
SELECT price  
FROM Sells  
WHERE bar = 'Joe''s Bar'  
      AND beer = 'Bud';
```

- Result of the Query:

price
2.50

Important Points

- ⌚ Two single quotes inside a string represent the single-quote (apostrophe).
- ⌚ Conditions in the WHERE clause can use AND, OR, NOT, and parentheses in the usual way boolean conditions are built.
- ⌚ SQL is *case-insensitive*. In general, upper and lower case characters are the same, **except inside quoted strings**.

Patterns

- ⌚ WHERE clauses can have conditions in which a string is compared with a pattern, to see if it matches.
- ⌚ General form:
 - ⌚ <Attribute> LIKE <pattern>
 - ⌚ <Attribute> NOT LIKE <pattern>
- ⌚ Pattern is a quoted string
 - ⌚ with % = “any string”
 - ⌚ with _ = “any character.”

Example

- From Drinkers(name, addr, phone) find the drinkers with exchange 555:

```
SELECT name  
FROM Drinkers  
WHERE phone LIKE '%555-_____';
```

NULL Values

- ⌚ Tuples in SQL relations can have NULL as a value for one or more components.
- ⌚ Meaning depends on context. Two common cases:
 - ⌚ *Missing value* : e.g., we know Joe's Bar has some address, but we don't know what it is.
 - ⌚ *Inapplicable* : e.g., the value of attribute `spouse` for an unmarried person.

Comparing NULL's to Values

- The logic of conditions in SQL is really 3-valued logic: TRUE, FALSE, UNKNOWN.
- When any value is compared with NULL, the truth value is UNKNOWN.
- But a query only produces a tuple in the answer if its truth value for the WHERE clause is TRUE (not FALSE or UNKNOWN).

Three-Valued Logic

- ☞ To understand how AND, OR, and NOT work in 3-valued logic, think of TRUE = 1, FALSE = 0, and UNKNOWN = $\frac{1}{2}$.
- ☞ AND = MIN; OR = MAX, $\text{NOT}(x) = 1-x$.
- ☞ Example:

TRUE AND (FALSE OR NOT(UNKNOWN)) =
 $\text{MIN}(1, \text{MAX}(0, (1 - \frac{1}{2}))) =$
 $\text{MIN}(1, \text{MAX}(0, \frac{1}{2})) = \text{MIN}(1, \frac{1}{2}) = \frac{1}{2}$.

Surprising Example

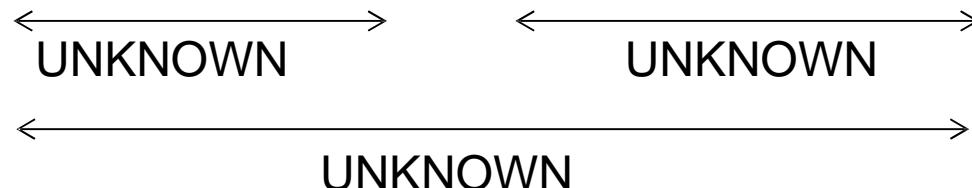
- From the following Sells relation:

bar	beer	price
Joe's Bar	Bud	NULL

```
SELECT bar
```

```
FROM Sells
```

```
WHERE price < 2.00 OR price >= 2.00;
```



Multirelation Queries

- ☞ Interesting queries often combine data from more than one relation.
- ☞ We can address several relations in one query by listing them all in the FROM clause.
- ☞ Distinguish attributes of the same name by “<relation>.<attribute>”

Example

- Using relations Likes(drinker, beer) and Frequent(drinker, bar), find the beers liked by at least one person who frequents Joe's Bar.

```
SELECT beer  
FROM Likes, Frequent  
WHERE bar = 'Joe''s Bar' AND  
Frequent.drinker = Likes.drinker;
```

Formal Semantics

- ⌚ Almost the same as for single-relation queries:
 1. Start with the product of all the relations in the FROM clause.
 2. Apply the selection condition from the WHERE clause.
 3. Project onto the list of attributes and expressions in the SELECT clause.

Operational Semantics

- ⌚ Imagine one tuple-variable for each relation in the FROM clause.
- ⌚ These tuple-variables visit each combination of tuples, one from each relation.
- ⌚ If the tuple-variables are pointing to tuples that satisfy the WHERE clause, send these tuples to the SELECT clause.

Example

drinker	bar
Sally	Joe's

check these
are equal

check for
Joe's

drinker	beer
Sally	Bud

output

Explicit Tuple-Variables

- ⌚ Sometimes, a query needs to use two copies of the same relation.
- ⌚ Distinguish copies by following the relation name by the name of a tuple-variable, in the FROM clause.
- ⌚ It's always an option to rename relations this way, even when not essential.

Example: Self-Join

- From Beers(name, manf), find all pairs of beers by the same manufacturer.

Do not produce pairs like (Bud, Bud).

Produce pairs in alphabetic order, e.g. (Bud, Miller), not (Miller, Bud).

```
SELECT b1.name, b2.name  
FROM Beers b1, Beers b2  
WHERE b1.manf = b2.manf  
      AND b1.name < b2.name;
```

Union, Intersection, and Difference

- Union, intersection, and difference of relations are expressed by the following forms, each involving subqueries:
 - (subquery) UNION (subquery)
 - (subquery) INTERSECT (subquery)
 - (subquery) [EXCEPT | MINUS](subquery)

Example

- ⌚ From relations Likes(drinker, beer), Sells(bar, beer, price) and Frequents(drinker, bar), find the drinkers and beers such that:
 1. The drinker likes the beer, and
 2. The drinker frequents at least one bar that sells the beer.

Solution

Notice trick:
subquery is
really a stored
table.

(SELECT * FROM Likes)

INTERSECT

(SELECT drinker, beer
FROM Sells, Frequents
WHERE Frequents.bar = Sells.bar
);

The drinker frequents
a bar that sells the
beer.

Bag (multiset) Semantics

- ☞ Although the SELECT-FROM-WHERE statement uses bag semantics, the default for union, intersection, and difference is set semantics.
- ☞ That is, duplicates are eliminated as the operation is applied.

Motivation: Efficiency

- ⌚ When doing projection in relational algebra, it is easier to avoid eliminating duplicates.
 - ⌚ Just work tuple-at-a-time.
-
- ⌚ When doing intersection or difference, it is most efficient to sort the relations first.
 - ⌚ At that point you may as well eliminate the duplicates anyway.

Controlling Duplicate Elimination

- ☞ Force the result to be a set by

SELECT DISTINCT . . .

- ☞ Force the result to be a bag (i.e., don't eliminate duplicates) by ALL, as in . . .

UNION ALL . . .

Example: DISTINCT

- From Sells(bar, beer, price), find all the different prices charged for beers:

```
SELECT DISTINCT price  
FROM Sells;
```

- Notice that without DISTINCT, each price would be listed as many times as there were bar/beer pairs at that price.

Example: ALL

- Using relations Frequent(drinker, bar) and Likes(drinker, beer):

```
(SELECT drinker FROM Frequent)
```

```
EXCEPT ALL
```

```
(SELECT drinker FROM Likes);
```

- Lists drinkers who frequent more bars than they like beers, and does so as many times as the difference of those counts.

Join Expressions

- ⌚ SQL provides a number of expression forms that act like varieties of join in relational algebra.
 - ⌚ But using bag semantics, not set semantics.
-
- ⌚ These expressions can be stand-alone queries or used in place of relations in a FROM clause.

Products and Natural Joins

⌚ Natural join:

R NATURAL JOIN S;

⌚ Product:

R CROSS JOIN S;

⌚ Example:

Likes NATURAL JOIN Sells;

⌚ Relations can be parenthesized subqueries, as well.

Theta Join

- R JOIN S ON<condition>
- Example: using `Drinkers(name, addr)` and `Frequents(drinker, bar)`:

```
Drinkers JOIN Frequents ON  
name = drinker;
```

gives us all (d, a, d, b) quadruples such that drinker d lives at address a and frequents bar b .

Subqueries

- A parenthesized SELECT-FROM-WHERE statement (*subquery*) can be used as a value in a number of places, including FROM and WHERE clauses.
- Example: in place of a relation in the FROM clause, we can place another query, and then query its result.
- Better use a tuple-variable to name tuples of the result.

Subqueries That Return One Tuple

- ⌚ If a subquery is guaranteed to produce one tuple, then the subquery can be used as a value.
- ⌚ Usually, the tuple has one component.
- ⌚ Also typically, a single tuple is guaranteed by keyness of attributes.
- ⌚ A run-time error occurs if there is no tuple or more than one tuple.

Example: Single-Tuple Subquery

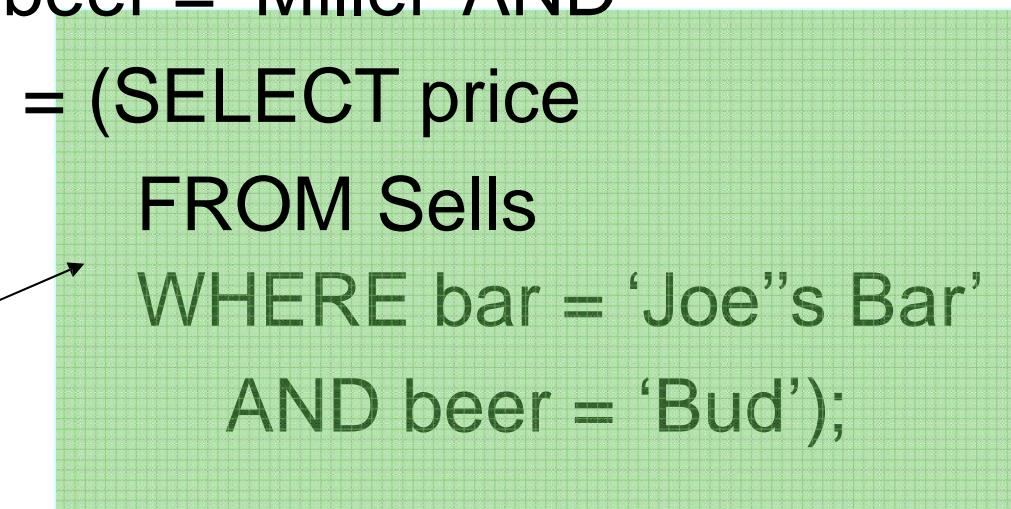
- ⌚ From Sells(bar, beer, price), find the bars that serve Miller for the same price Joe charges for Bud.

- ⌚ Two queries would surely work:
 1. Find the price Joe charges for Bud.
 2. Find the bars that serve Miller at that price.

Query + Subquery Solution

```
SELECT bar  
FROM Sells  
WHERE beer = 'Miller' AND  
      price = (SELECT price  
                FROM Sells  
                WHERE bar = 'Joe''s Bar'  
                  AND beer = 'Bud');
```

The price at
which Joe
sells Bud



The IN Operator

- $\langle \text{tuple} \rangle \text{ IN } \langle \text{relation} \rangle$ is true if and only if the tuple is a member of the relation.
- $\langle \text{tuple} \rangle \text{ NOT IN } \langle \text{relation} \rangle$ means the opposite.
- IN-expressions can appear in WHERE clauses.
- The $\langle \text{relation} \rangle$ is often a subquery.

Example: IN

- From Beers(name, manf) and Likes(drinker, beer), find the name and manufacturer of each beer that Fred likes.

```
SELECT *
```

```
FROM Beers
```

```
WHERE name IN (SELECT beer
```

The set of
beers Fred
likes

```
FROM Likes  
WHERE drinker = 'Fred');
```

The Exists Operator

- EXIST(<relation>) is true if and only if the <relation> is not empty.
- Being a boolean-valued operator, EXISTS can appear in WHERE clauses.
- Example: From Beers(name, manf), find those beers that are the unique beer by their manufacturer.

Example Query with EXISTS

```
SELECT name  
FROM Beers b1  
  
WHERE NOT EXISTS(
```

Set of beers with the same manf as b1, but not the same beer

Notice scope rule: manf refers to closest nested FROM with a relation having that attribute.

```
SELECT *  
FROM Beers  
WHERE manf = b1.manf AND  
      name <> b1.name);
```

Notice the SQL “not equals” operator

The Operator ANY

- $x = \text{ANY}(\text{ <relation> })$ is a boolean condition meaning that x equals at least one tuple in the relation.
- Similarly, $=$ can be replaced by any of the comparison operators.
- Example: $x \geq \text{ANY}(\text{ <relation> })$ means x is not smaller than all tuples in the relation.
- Note tuples must have one component only.

The Operator ALL

- ⌚ Similarly, $x <> \text{ALL}(\langle \text{relation} \rangle)$ is true if and only if for every tuple t in the relation, x is not equal to t .
 - ⌚ That is, x is not a member of the relation.
- ⌚ The $<>$ can be replaced by any comparison operator.
- ⌚ Example: $x \geq \text{ALL}(\langle \text{relation} \rangle)$ means there is no tuple larger than x in the relation.

Example: ALL

- c From Sells(bar, beer, price), find the beer(s) sold for the highest price.

```
SELECT beer  
FROM Sells  
WHERE price >= ALL(  
    SELECT price  
    FROM Sells);
```

price from the outer
Sells must not be
less than any price.

Aggregation Operators

- ☞ Aggregation operators are not operators of relational algebra.
- ☞ Rather, they apply to entire columns of a table and produce a single result.
- ☞ The most important examples: SUM, AVG, COUNT, MIN, and MAX.

Aggregations

- SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause to produce that aggregation on the column.
- Also, COUNT(*) counts the number of tuples.

Example: Aggregation

- From Sells(bar, beer, price), find the average price of Bud:

```
SELECT AVG(price)  
FROM Sells  
WHERE beer = 'Bud' ;
```

Eliminating Duplicates in an Aggregation

- ⌚ DISTINCT inside an aggregation causes duplicates to be eliminated before the aggregation.
- ⌚ Example: find the number of different prices charged for Bud:

```
SELECT COUNT(DISTINCT price)
FROM Sells
WHERE beer = 'Bud' ;
```

NULL's Ignored in Aggregation

- NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column.
- But if there are no non-NULL values in a column, then the result of the aggregation is NULL.

Example: Effect of NULL's

```
SELECT count(*)  
FROM Sells  
WHERE beer = 'Bud';
```

The number of bars
that sell Bud.

```
SELECT count(price)  
FROM Sells  
WHERE beer = 'Bud';
```

The number of bars
that sell Bud at a
known price.

Grouping

- We may follow a SELECT-FROM-WHERE expression by GROUP BY and a list of attributes.
- The relation that results from the SELECT-FROM-WHERE is grouped according to the values of all those attributes, and any aggregation is applied only within each group.

Example: Grouping

- c From Sells(bar, beer, price), find the average price for each beer:

```
SELECT beer, AVG(price)  
FROM Sells  
GROUP BY beer;
```

Example: Grouping

- c From Sells(bar, beer, price) and Frequent(drinker, bar), find for each drinker the average price of Bud at the bars they frequent:

```
SELECT drinker, AVG(price)
```

```
FROM Frequent, Sells
```

```
WHERE beer = 'Bud' AND
```

```
    Frequent.bar = Sells.bar
```

```
GROUP BY drinker;
```

Compute
drinker-bar-
price of Bud
tuples first,
then group
by drinker.

Restriction on SELECT Lists With Aggregation

- ☞ If any aggregation is used, then each element of the SELECT list must be either:
 1. Aggregated, or
 2. An attribute on the GROUP BY list.

Illegal Query Example

- ☞ You might think you could find the bar that sells Bud the cheapest by:

```
SELECT bar, MIN(price)  
FROM Sells  
WHERE beer = 'Bud';
```

- ☞ But this query is illegal in SQL.
 - ☞ Why? Note bar is neither aggregated nor on the GROUP BY list.

HAVING Clauses

- ⌚ HAVING <condition> may follow a GROUP BY clause.
- ⌚ If so, the condition applies to each group, and groups not satisfying the condition are eliminated.

Requirements on HAVING Conditions

- ⌚ These conditions may refer to any relation or tuple-variable in the FROM clause.
- ⌚ They may refer to attributes of those relations, as long as the attribute makes sense within a group; i.e., it is either:
 1. A grouping attribute, or
 2. Aggregated.

Example: HAVING

- c From Sells(bar, beer, price) and Beers(name, manf), find the average price of those beers that are either served in at least three bars or are manufactured by Pete's.

Solution

```
SELECT beer, AVG(price)  
FROM Sells  
GROUP BY beer
```

```
HAVING COUNT(bar) >= 3 OR  
beer IN (SELECT name  
FROM Beers  
WHERE manf = 'Pete''s');
```

Beer groups with at least 3 non-NULL bars and also beer groups where the manufacturer is Pete's.

Beers manufactured by Pete's.

Data models and Databases

Extended Relational Algebra

Relational Algebra

- ⌚ What is an “Algebra”?
- ⌚ Mathematical system consisting of:
 - ⌚ *Operands* --- variables or values from which new values can be constructed.
 - ⌚ *Operators* --- symbols denoting procedures that construct new values from given values.

Core Relational Algebra

- ⌚ Union, intersection, and difference.
 - ⌚ Usual set operations, but require both operands have the same relation schema.
- ⌚ Selection: picking certain rows.
- ⌚ Projection: picking certain columns.
- ⌚ Products and joins: compositions of relations.
- ⌚ Renaming of relations and attributes.

Relational Algebra on Bags

- ⌚ A *bag* is like a set, but an element may appear more than once.
 - ⌚ *Multiset* is another name for “bag.”
-
- ⌚ Example: $\{1,2,1,3\}$ is a bag. $\{1,2,3\}$ is also a bag that happens to be a set.
 - ⌚ Bags also resemble lists, but order in a bag is unimportant.
 - ⌚ Example: $\{1,2,1\} = \{1,1,2\}$ as bags, but $[1,2,1] \neq [1,1,2]$ as lists.

Why Bags?

- SQL, the most important query language for relational databases is actually a bag language.
- SQL will eliminate duplicates, but usually only if you ask it to do so explicitly.
- Some operations, like projection, are much more efficient on bags than sets.

Operations on Bags

- Selection applies to each tuple, so its effect on bags is like its effect on sets.
- Projection also applies to each tuple, but as a bag operator, we **do not eliminate duplicates**.
- Products and joins are done on each pair of tuples, so duplicates in bags have no effect on how we operate.

Example: Bag Selection

R

A	B
1	2
5	6
1	2

S

B	C
3	4
7	8

SELECT _{$A+B < 5$} (R) =

A	B
1	2
1	2

Example: Bag Projection

R

A	B
1	2
5	6
1	2

S

B	C
3	4
7	8

$\text{PROJECT}_A(R) =$

A
1
5
1

Example: Bag Product

R

A	B
1	2
5	6
1	2

S

B	C
3	4
7	8

$R * S =$

A	R.B	S.B	C
1	2	3	4
1	2	7	8
5	6	3	4
5	6	7	8
1	2	3	4
1	2	7	8

Example: Bag Theta-Join

R

A	B
1	2
5	6
1	2

S

B	C
3	4
7	8

$R \text{ JOIN }_{R.B < S.B} S =$

A	R.B	S.B	C
1	2	3	4
1	2	7	8
5	6	7	8
1	2	3	4
1	2	7	8

Bag Union

- Union, intersection, and difference need new definitions for bags.
- An element appears in the union of two bags the sum of the number of times it appears in each bag.
- Example: $\{1,2,1\} \text{ UNION } \{1,1,2,3,1\} = \{1,1,1,1,1,2,2,3\}$

Bag Intersection

- An element appears in the intersection of two bags the minimum of the number of times it appears in either.
- Example: $\{1,1,2,1\} \text{ INTER } \{1,1,2,3\} = \{1,1,2\}$.

Bag Difference

- ⌚ An element appears in the difference $A - B$ of bags as many times as it appears in A , minus the number of times it appears in B .
 - ⌚ But never less than 0 times.
- ⌚ Example: $\{1,2,1\} - \{1,2,3\} = \{1\}$.

Beware: Bag Laws <> Set Laws

- ⌚ Not all algebraic laws that hold for sets also hold for bags.
- ⌚ For one example, the commutative law for union ($R \text{ UNION } S = S \text{ UNION } R$) does hold for bags.
- ⌚ Since addition is commutative, adding the number of times x appears in R and S doesn't depend on the order of R and S .

An Example of Inequivalence

- ☞ Set union is *idempotent*, meaning that
 $S \text{ UNION } S = S$.
- ☞ However, for bags, if x appears n times in S ,
then it appears $2n$ times in $S \text{ UNION } S$.
- ☞ Thus $S \text{ UNION } S <> S$ in general.

The Extended Algebra

1. DELTA = eliminate duplicates from bags.
2. TAU = sort tuples.
3. *Extended projection* : arithmetic, duplication of columns.
4. GAMMA = grouping and aggregation.
5. OUTERJOIN: avoids “dangling tuples” = tuples that do not join with anything.

Duplicate Elimination

- $R1 := \text{DELTA}(R2)$.
- $R1$ consists of one copy of each tuple that appears in $R2$ one or more times.

Example: Duplicate Elimination

$R =$

A	B
1	2
3	4
1	2

$\Delta R =$

A	B
1	2
3	4

Sorting

- ⌚ $R1 := \text{TAU}_L(R2)$.
- ⌚ L is a list of some of the attributes of $R2$.
- ⌚ $R1$ is the list of tuples of $R2$ sorted first on the value of the first attribute on L , then on the second attribute of L , and so on.
- ⌚ Break ties arbitrarily.
- ⌚ TAU is the only operator whose result is neither a set nor a bag.
- ⌚ ORDER BY in SQL

Example: Sorting

$R =$

A	B
1	2
3	4
5	2

$$\text{TAU}_B(R) = [(5,2), (1,2), (3,4)]$$

Extended Projection

- ☞ Using the same PROJ_L operator, we allow the list L to contain arbitrary expressions involving attributes, for example:
 1. Arithmetic on attributes, e.g., $A+B$.
 2. Duplicate occurrences of the same attribute.

Example: Extended Projection

$R =$

A	B
1	2
3	4

$\text{PROJ}_{A+B, A, A}(R) =$

A+B	A1	A2
3	1	1
7	3	3

Aggregation Operators

- ☞ Aggregation operators are not operators of relational algebra.
- ☞ Rather, they apply to entire columns of a table and produce a single result.
- ☞ The most important examples: SUM, AVG, COUNT, MIN, and MAX.

Example: Aggregation

R =

A	B
1	3
3	4
3	2

$$\text{SUM}(A) = 7$$

$$\text{COUNT}(A) = 3$$

$$\text{MAX}(B) = 4$$

$$\text{AVG}(B) = 3$$

Grouping Operator

- ☞ $R1 := \text{GAMMA}_L(R2)$. L is a list of elements that are either:
 1. Individual (*grouping*) attributes.
 2. $\text{AGG}(A)$, where AGG is one of the aggregation operators and A is an attribute.

Applying GAMMA_{L(R)}

- Group R according to all the grouping attributes on list L .
 - That is, form one group for each distinct list of values for those attributes in R .
- Within each group, compute AGG(A) for each aggregation on list L .
- Result has grouping attributes and aggregations as attributes. One tuple for each list of values for the grouping attributes and their group's aggregations.

Example: Grouping/Aggregation

$R =$

A	B	C
1	2	3
4	5	6
1	2	5

$$\text{GAMMA}_{A,B,\text{AVG}(C)}(R) = ??$$

First, group R :

A	B	C
1	2	3
1	2	5
4	5	6

Then, average C within groups:

A	B	AVG(C)
1	2	4
4	5	6

Outerjoin

- ⌚ Suppose we join $R \text{ JOIN}_C S$.
- ⌚ A tuple of R that has no tuple of S with which it joins is said to be *dangling*.
 - ⌚ Similarly for a tuple of S .
- ⌚ Outerjoin preserves dangling tuples by padding them with a special NULL symbol in the result.

Example: Outerjoin

R =

A	B
1	2
4	5

S =

B	C
2	3
6	7

(1,2) joins with (2,3), but the other two tuples
are dangling.

R OUTERJOIN S =

A	B	C
1	2	3
4	5	NULL
NULL	6	7



Data models and Databases



Database Modification

Database Modifications

- ⌚ A modification command does not return a result as a query does, but it changes the database in some way.
- ⌚ There are three kinds of modifications:
 1. *Insert* a tuple or tuples.
 2. *Delete* a tuple or tuples.
 3. *Update* the value(s) of an existing tuple or tuples.

Insertion

- To insert a single tuple: `INSERT INTO <relation> VALUES (<list of values>);`
- Example: add to Likes(drinker, beer) the fact that Sally likes Bud.

```
INSERT INTO Likes  
VALUES ('Sally', 'Bud');
```

Specifying Attributes in INSERT

- ☞ We may add to the relation name a list of attributes.
- ☞ There are two reasons to do so:
 1. We forget the standard order of attributes for the relation.
 2. We don't have values for all attributes, and we want the system to fill in missing components with NULL or a default value.

Example: Specifying Attributes

- Another way to add the fact that Sally likes Bud to Likes(drinker, beer):

```
INSERT INTO Likes(beer, drinker)  
VALUES ('Bud', 'Sally');
```

Inserting Many Tuples

- We may insert the entire result of a query into a relation, using the form:

```
INSERT INTO <relation>
( <subquery> );
```

Example: Insert a Subquery

- Using `Frequents(drinker, bar)`, enter into the new relation `PotBuddies(name)` all of Sally's "potential buddies," i.e., those drinkers who frequent at least one bar that Sally also frequents.

Solution

The other
drinker

```
INSERT INTO PotBuddies
(SELECT d2.drinker
FROM Frequent d1, Frequent d2
WHERE d1.drinker = 'Sally' AND
d2.drinker <> 'Sally' AND
d1.bar = d2.bar
);
```

Pairs of Drinker
tuples where the
first is for Sally,
the second is for
someone else,
and the bars are
the same.

Deletion

- To delete tuples satisfying a condition from some relation:

```
DELETE FROM <relation>  
WHERE <condition>;
```

Example: Deletion

- Delete from Likes(drinker, beer) the fact that Sally likes Bud:

```
DELETE FROM Likes  
WHERE drinker = 'Sally' AND  
beer = 'Bud';
```

Example: Delete all Tuples

- ⌚ Make the relation Likes empty:

```
DELETE FROM Likes;
```

- ⌚ Note no WHERE clause needed.

Example: Delete Many Tuples

- >Delete from Beers(name, manf) all beers for which there is another beer by the same manufacturer.

```
DELETE FROM Beers b  
WHERE EXISTS (
```

```
    SELECT name FROM Beers  
    WHERE manf = b.manf AND  
          name <> b.name);
```

Beers with the same manufacturer and a different name from the name of the beer represented by tuple b.

Semantics of Deletion -- 1

- ⌚ Suppose Anheuser-Busch makes only Bud and Bud Lite.
- ⌚ Suppose we come to the tuple b for Bud first.
- ⌚ The subquery is nonempty, because of the Bud Lite tuple, so we delete Bud.
- ⌚ Now, When b is the tuple for Bud Lite, do we delete that tuple too?

Semantics of Deletion -- 2

- ⌚ The answer is that we *do* delete Bud Lite as well.
- ⌚ The reason is that deletion proceeds in two stages:
 1. Mark all tuples for which the WHERE condition is satisfied in the original relation.
 2. Delete the marked tuples.

Updates

- To change certain attributes in certain tuples of a relation:

UPDATE <relation>

SET <list of attribute assignments>

WHERE <condition on tuples>;

Example: Update

- Change drinker Fred's phone number to 555-1212:

```
UPDATE Drinkers  
SET phone = '555-1212'  
WHERE name = 'Fred';
```

Example: Update Several Tuples

⌚ Make \$4 the maximum price for beer:

```
UPDATE Sells  
SET price = 4.00  
WHERE price > 4.00;
```

Entity-Relationship Model

Jeffrey D. Ullman

Purpose of E/R Model

- ◆ The E/R model allows us to sketch database schema designs.
 - ◆ Includes some constraints, but not operations.
- ◆ Designs are pictures called *entity-relationship diagrams*.
- ◆ Later: convert E/R designs to relational DB designs.

Framework for E/R

- ◆ Design is a serious business.
- ◆ The “boss” knows they want a database, but they don’t know what they want in it.
- ◆ Sketching the key components is an efficient way to develop a working database.

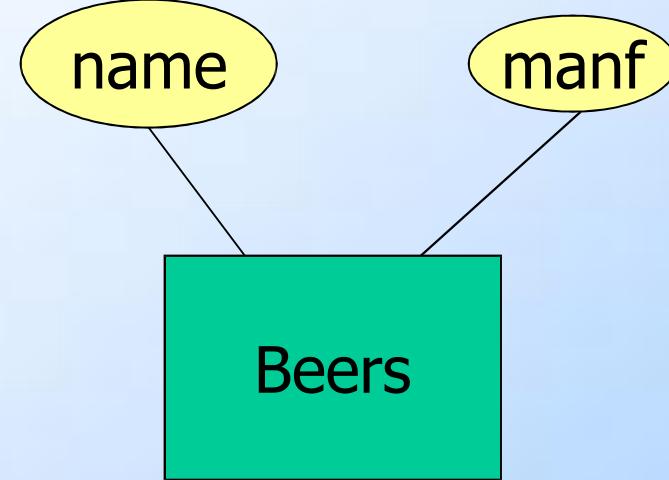
Entity Sets

- ◆ *Entity* = “thing” or object.
- ◆ *Entity set* = collection of similar entities.
 - ◆ Similar to a class in object-oriented languages.
- ◆ *Attribute* = property of (the entities of) an entity set.
 - ◆ Attributes are simple values, e.g. integers or character strings, not structs, sets, etc.

E/R Diagrams

- ◆ In an entity-relationship diagram:
 - ◆ Entity set = rectangle.
 - ◆ Attribute = oval, with a line to the rectangle representing its entity set.

Example:

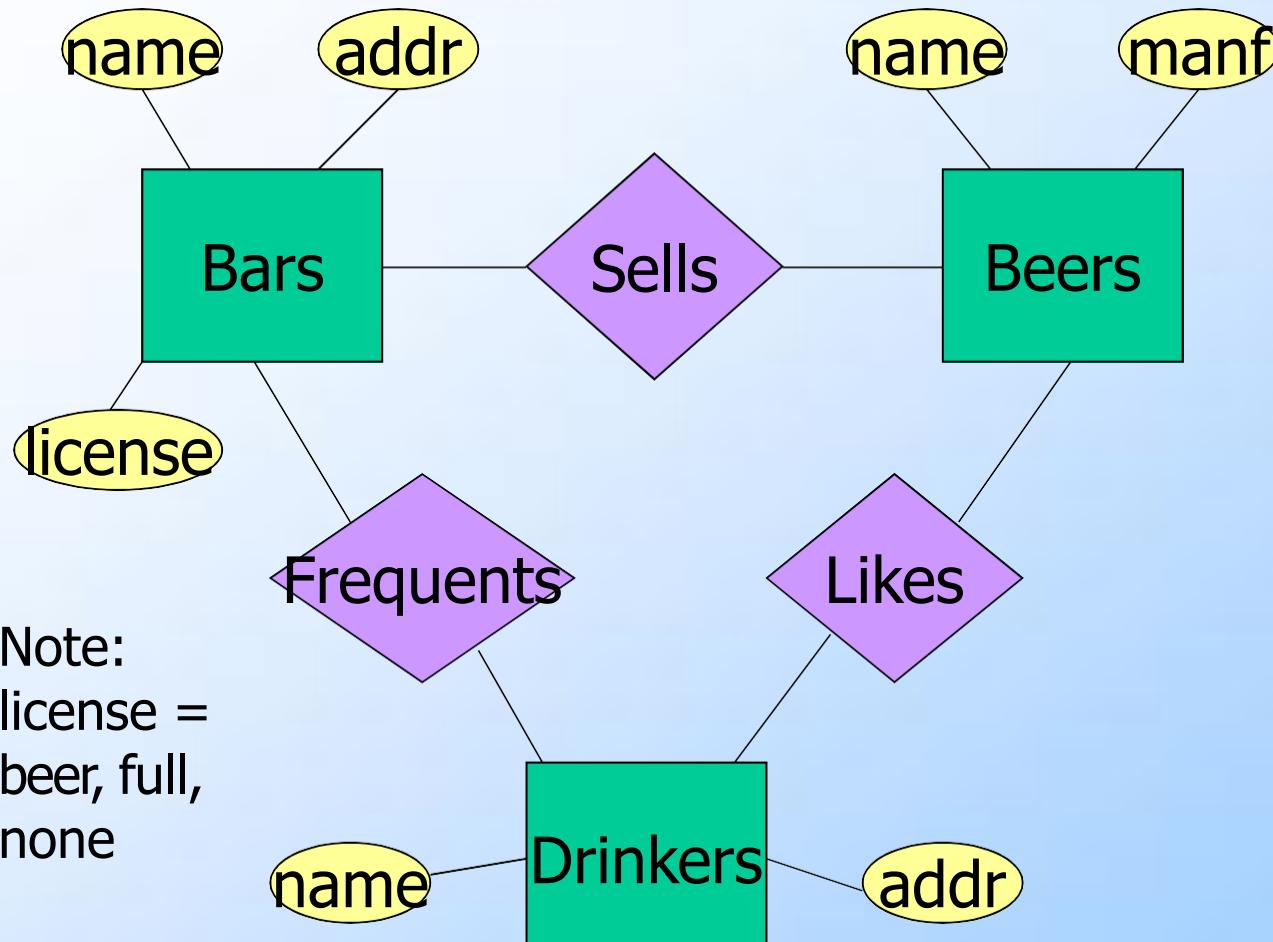


- ◆ Entity set **Beers** has two attributes, **name** and **manf** (manufacturer).
- ◆ Each **Beers** entity has values for these two attributes, e.g. (Bud, Anheuser-Busch)

Relationships

- ◆ A **relationship** connects two or more entity sets.
- ◆ It is represented by a diamond, with lines to each of the entity sets involved.

Example: Relationships



Note:

license =
beer, full,
none

Bars sell some beers.

Drinkers like some beers.

Drinkers frequent some bars.

Relationship Set

- ◆ The current “value” of an entity set is the set of entities that belong to it.
 - ◆ Example: the set of all bars in our database.
- ◆ The “value” of a relationship is a *relationship set*, a set of tuples with one component for each related entity set.

Example: Relationship Set

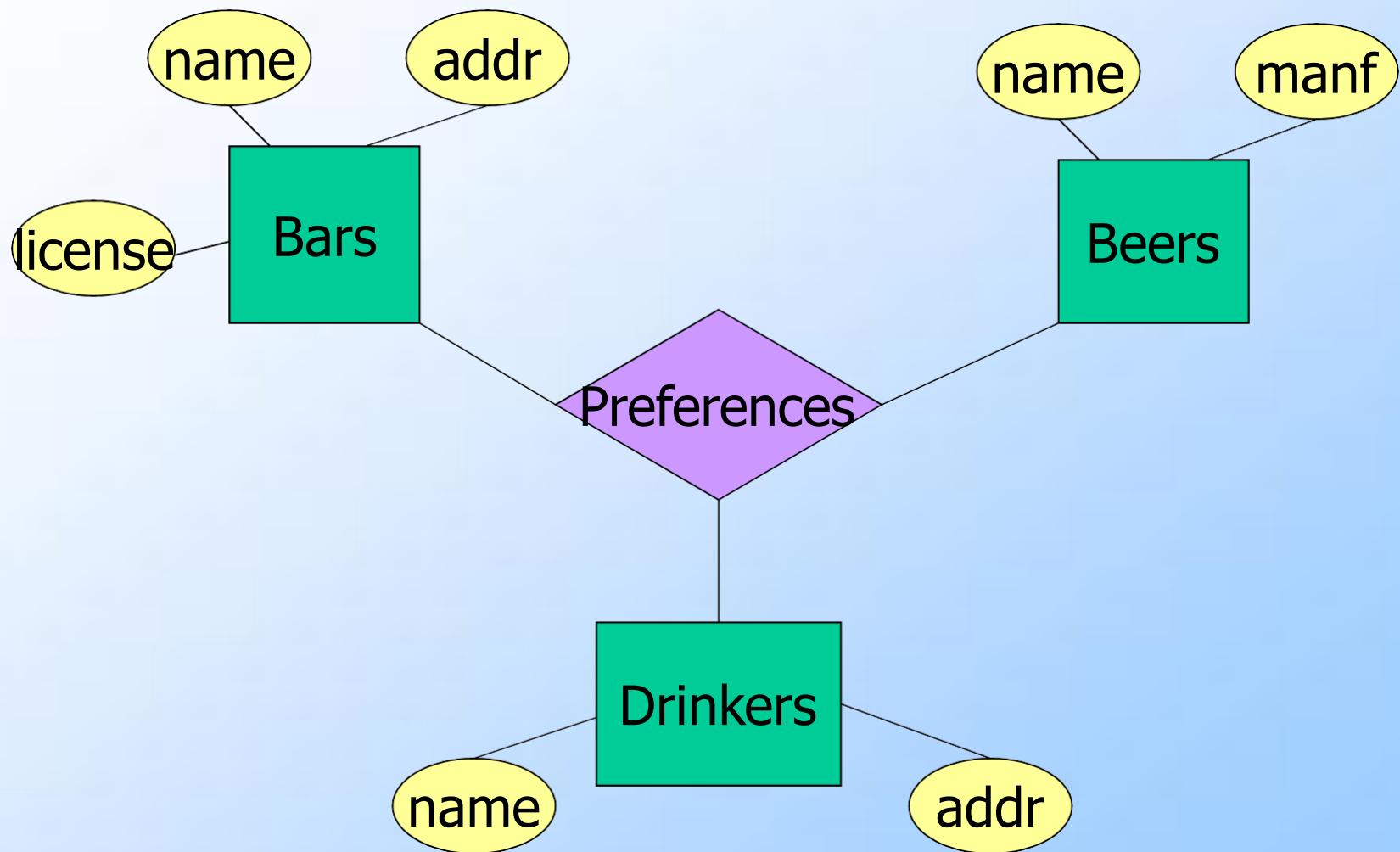
- ◆ For the relationship **Sells**, we might have a relationship set like:

Bar	Beer
Joe's Bar	Bud
Joe's Bar	Miller
Sue's Bar	Bud
Sue's Bar	Pete's Ale
Sue's Bar	Bud Lite

Multiway Relationships

- ◆ Sometimes, we need a relationship that connects more than two entity sets.
- ◆ Suppose that drinkers will only drink certain beers at certain bars.
 - ◆ Our three binary relationships **Likes**, **Sells**, and **Frequents** do not allow us to make this distinction.
 - ◆ But a 3-way relationship would.

Example: 3-Way Relationship



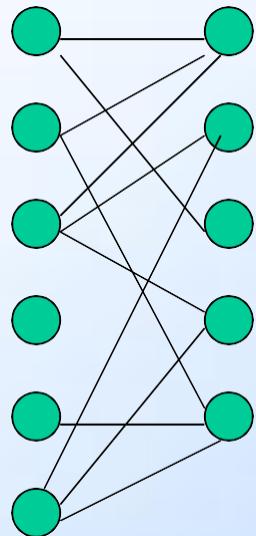
A Typical Relationship Set

Bar	Drinker	Beer
Joe's Bar	Ann	Miller
Sue's Bar	Ann	Bud
Sue's Bar	Ann	Pete's Ale
Joe's Bar	Bob	Bud
Joe's Bar	Bob	Miller
Joe's Bar	Cal	Miller
Sue's Bar	Cal	Bud Lite

Many-Many Relationships

- ◆ Focus: **binary** relationships, such as **Sells** between **Bars** and **Beers**.
- ◆ In a ***many-many* relationship**, an entity of either set can be connected to many entities of the other set.
 - ◆ E.g., a bar sells many beers; a beer is sold by many bars.

In Pictures:

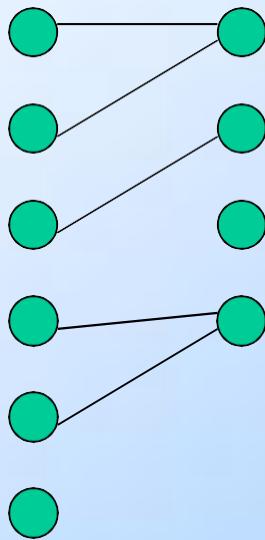


many-many

Many-One Relationships

- ◆ Some binary relationships are *many-one* from one entity set to another.
- ◆ Each entity of the first set is connected to at most one entity of the second set.
- ◆ But an entity of the second set can be connected to zero, one, or many entities of the first set.

In Pictures:



many-one

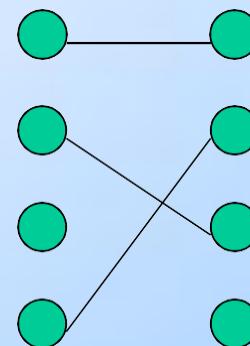
Example: Many-One Relationship

- ◆ **Favorite**, from **Drinkers** to **Beers** is many-one.
 - ◆ A drinker has at most one favorite beer.
 - ◆ But a beer can be the favorite of any number of drinkers, including zero.

One-One Relationships

- ◆ In a *one-one relationship*, each entity of either entity set is related to at most one entity of the other set.
- ◆ Example: Relationship **Best-seller** between entity sets **Manfs** (manufacturer) and **Beers**.
 - ◆ A beer cannot be made by more than one manufacturer, and no manufacturer can have more than one best-seller (assume no ties).

In Pictures:

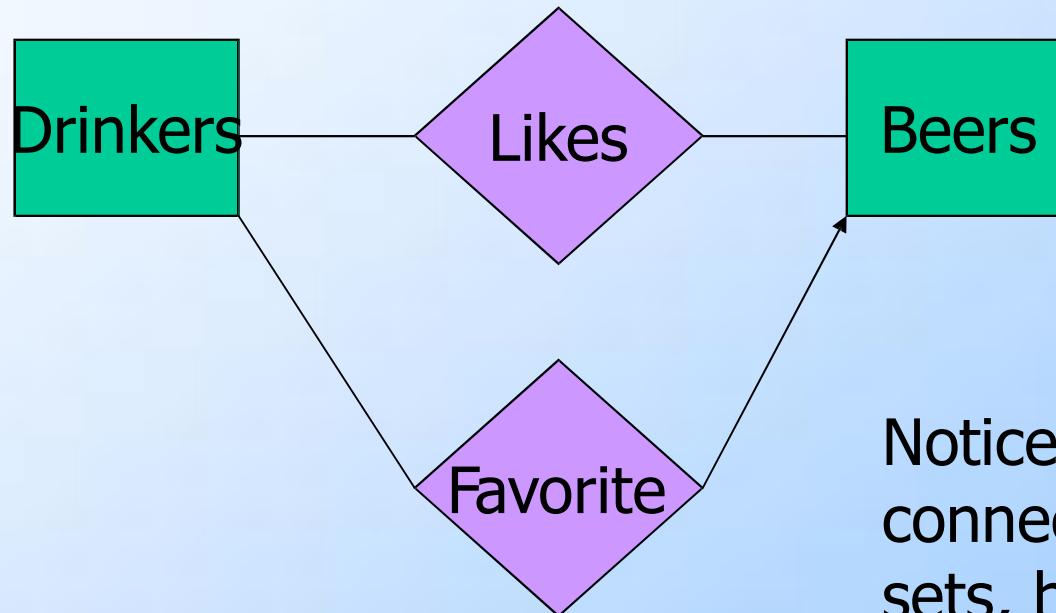


one-one

Representing “Multiplicity”

- ◆ Show a many-one relationship by an arrow entering the “one” side.
 - ◆ Remember: Like a functional dependency.
- ◆ Show a one-one relationship by arrows entering both entity sets.
- ◆ Rounded arrow = “exactly one,” i.e., each entity of the first set is related to exactly one entity of the target set.

Example: Many-One Relationship



Notice: two relationships connect the same entity sets, but are different.

Example: One-One Relationship

- ◆ Consider **Best-seller** between **Manfs** and **Beers**.
- ◆ Some beers are not the best-seller of any manufacturer, so a rounded arrow to **Manfs** would be inappropriate.
- ◆ But a beer manufacturer has to have a best-seller.

In the E/R Diagram



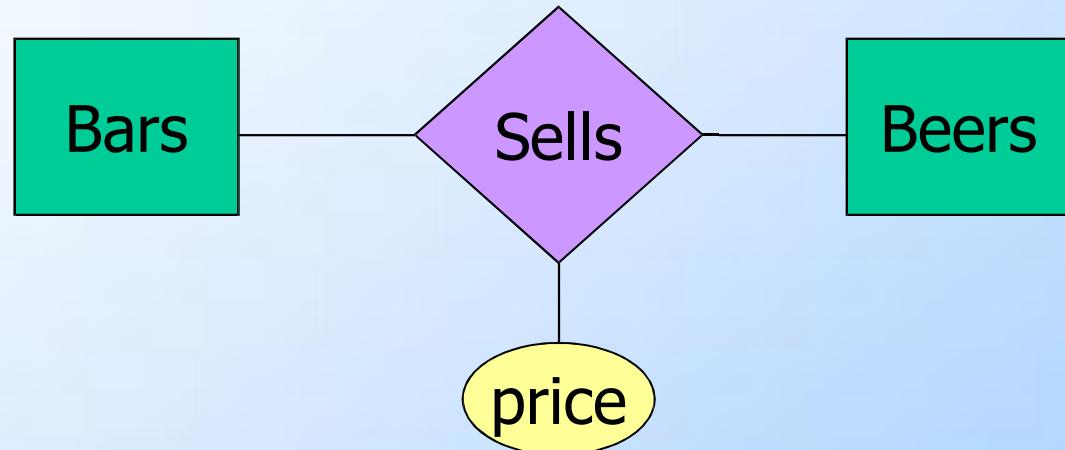
A beer is the best-seller for 0 or 1 manufacturer.

A manufacturer has exactly one best seller.

Attributes on Relationships

- ◆ Sometimes it is useful to attach an attribute to a relationship.
- ◆ Think of this attribute as a property of tuples in the relationship set.

Example: Attribute on Relationship

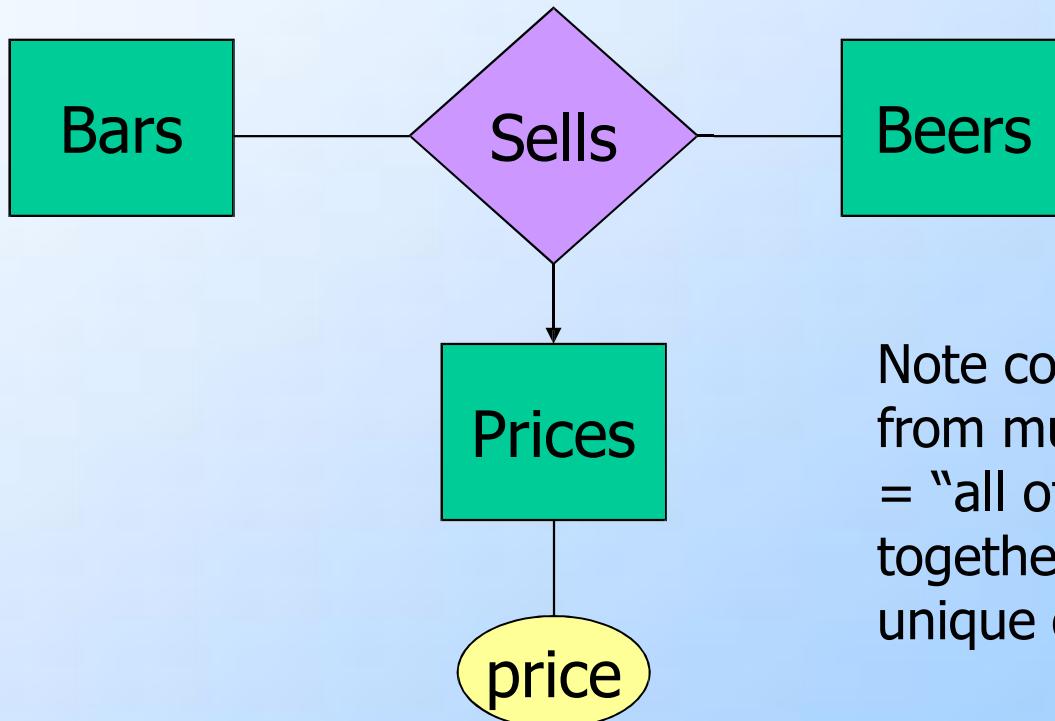


Price is a function of both the bar and the beer, not of one alone.

Equivalent Diagrams Without Attributes on Relationships

- ◆ Create an entity set representing values of the attribute.
- ◆ Make that entity set participate in the relationship.

Example: Removing an Attribute from a Relationship



Note convention: arrow from multiway relationship = “all other entity sets together determine a unique one of these.”

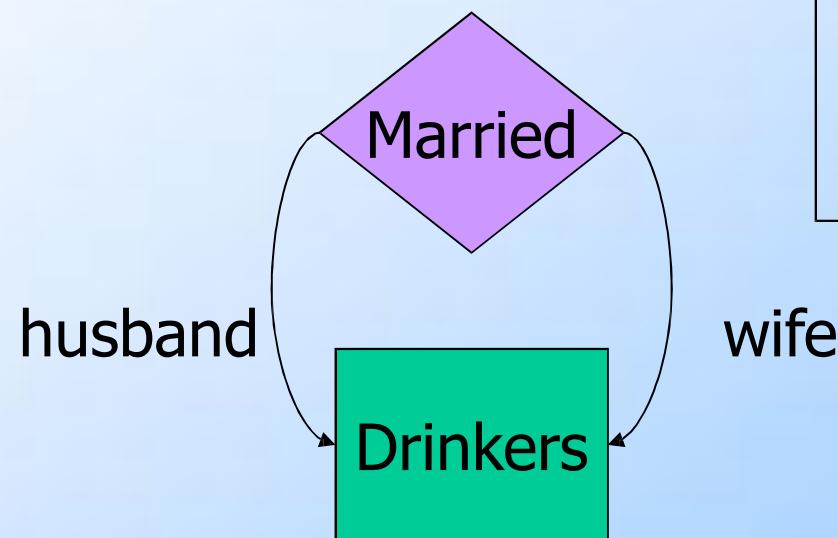
Roles

- ◆ Sometimes an entity set appears more than once in a relationship.
- ◆ Label the edges between the relationship and the entity set with names called *roles*.

Example: Roles

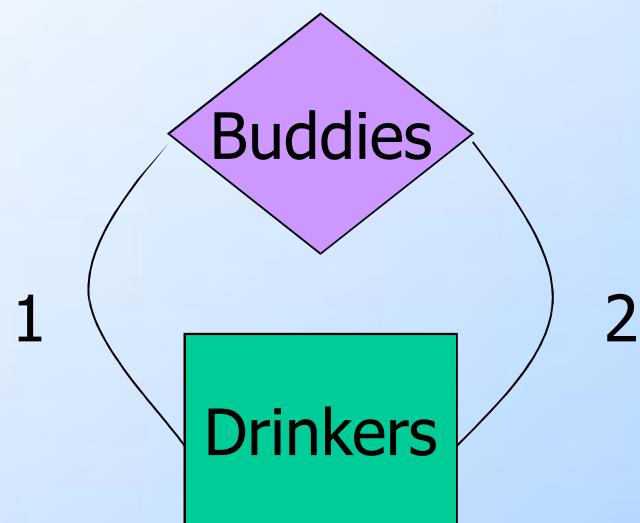
Relationship Set

Husband	Wife
Bob	Ann
Joe	Sue
...	...



Example: Roles

Relationship Set



Buddy1	Buddy2
Bob	Ann
Joe	Sue
Ann	Bob
Joe	Moe
...	...

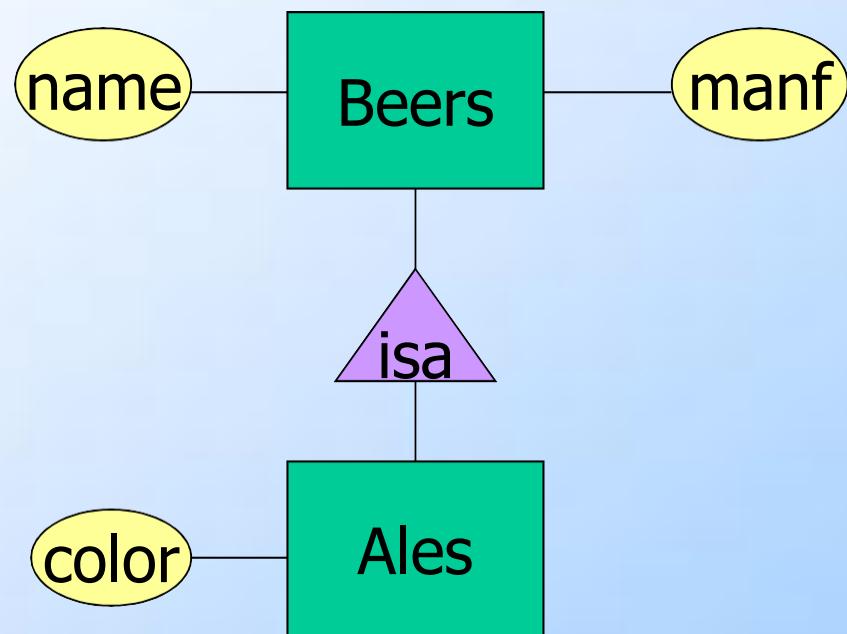
Subclasses

- ◆ *Subclass* = special case = fewer entities = more properties.
- ◆ **Example:** Ales are a kind of beer.
 - ◆ Not every beer is an ale, but some are.
 - ◆ Let us suppose that in addition to all the *properties* (attributes and relationships) of beers, ales also have the attribute **color**.

Subclasses in E/R Diagrams

- ◆ Assume subclasses form a tree.
 - ◆ I.e., no multiple inheritance.
- ◆ Isa triangles indicate the subclass relationship.
 - ◆ Point to the superclass.

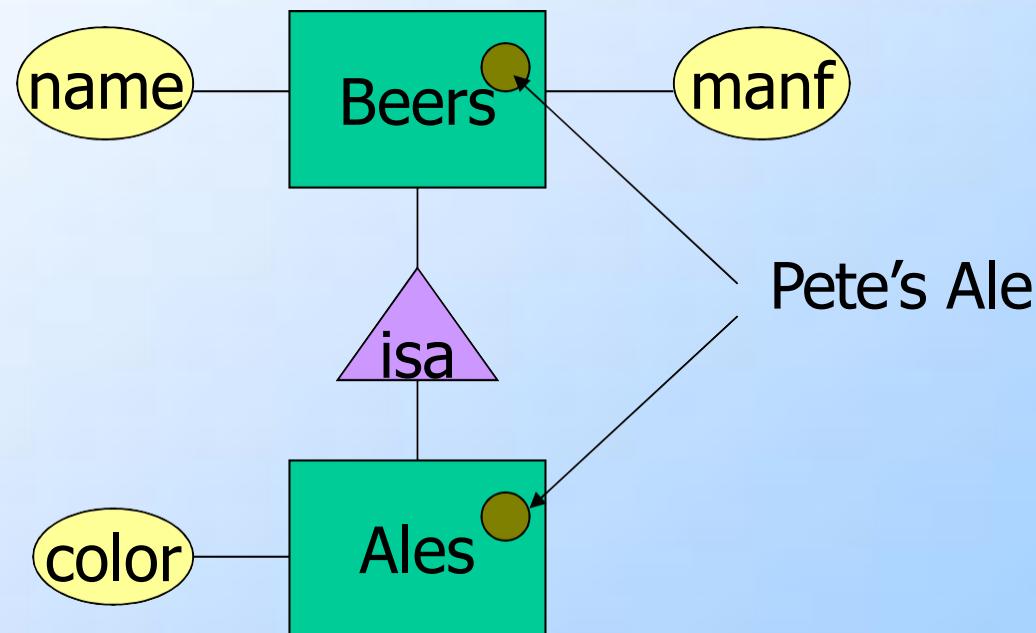
Example: Subclasses



E/R Vs. Object-Oriented Subclasses

- ◆ In OO, objects are in one class only.
 - ◆ Subclasses inherit from superclasses.
- ◆ In contrast, E/R entities have *representatives* in all subclasses to which they belong.
 - ◆ **Rule:** if entity e is represented in a subclass, then e is represented in the superclass (and recursively up the tree).

Example: Representatives of Entities



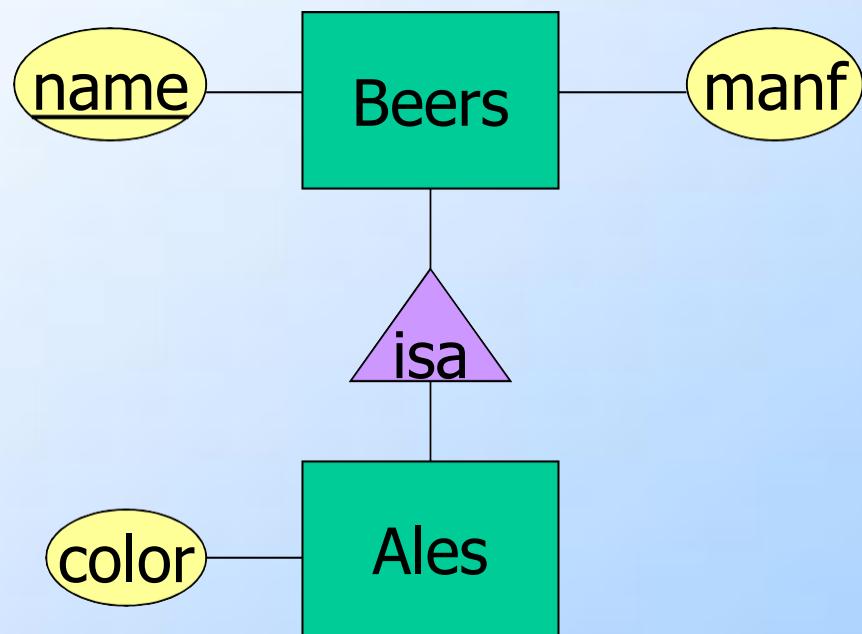
Keys

- ◆ A *key* is a set of attributes for one entity set such that no two entities in this set agree on all the attributes of the key.
 - ◆ It is allowed for two entities to agree on some, but not all, of the key attributes.
- ◆ We must designate a key for every entity set.

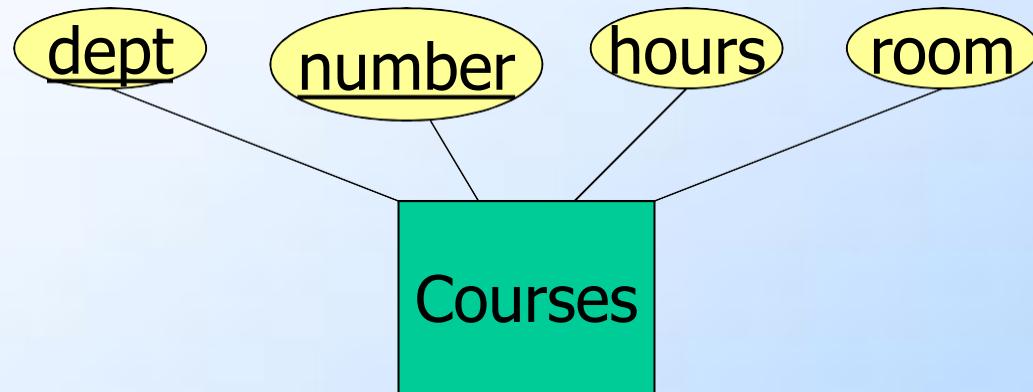
Keys in E/R Diagrams

- ◆ Underline the key attribute(s).
- ◆ In an Isa hierarchy, only the root entity set has a key, and it must serve as the key for all entities in the hierarchy.

Example: name is Key for Beers



Example: a Multi-attribute Key



- Note that **hours** and **room** could also serve as a key, but we must select only one key.

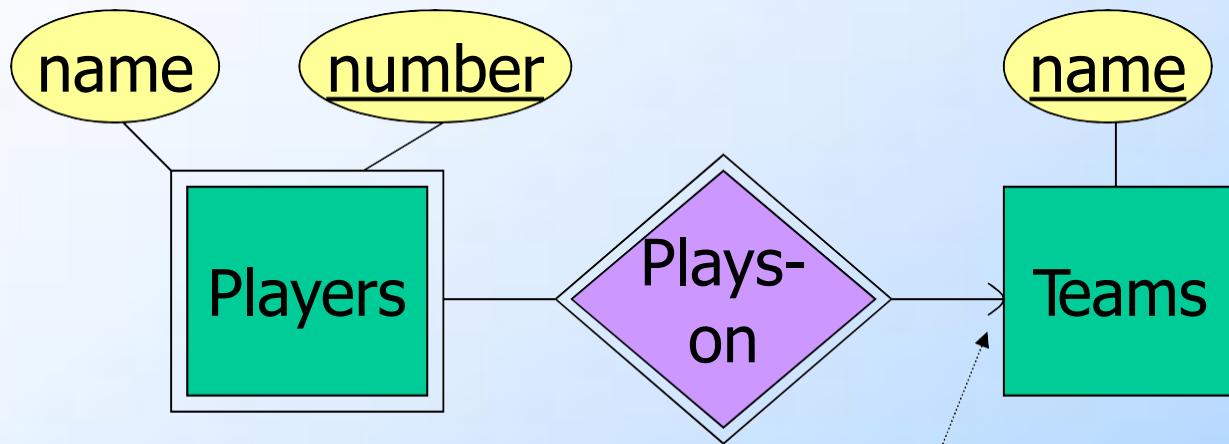
Weak Entity Sets

- ◆ Occasionally, entities of an entity set need “help” to identify them uniquely.
- ◆ Entity set E is said to be *weak* if in order to identify entities of E uniquely, we need to follow one or more many-one relationships from E and include the key of the related entities from the connected entity sets.

Example: Weak Entity Set

- ◆ **name** is almost a key for football players, but there might be two with the same name.
- ◆ **number** is certainly not a key, since players on two teams could have the same number.
- ◆ But **number**, together with the team **name** related to the player by **Plays-on** should be unique.

In E/R Diagrams



Note: must be rounded
because each player needs
a team to help with the key.

- Double diamond for *supporting* many-one relationship.
- Double rectangle for the weak entity set.

Weak Entity-Set Rules

- ◆ A weak entity set has one or more many-one relationships to other (supporting) entity sets.
 - ◆ Not every many-one relationship from a weak entity set need be supporting.
 - ◆ But supporting relationships must have a rounded arrow (entity at the “one” end is guaranteed).

Weak Entity-Set Rules – (2)

- ◆ The key for a weak entity set is its own underlined attributes and the keys for the supporting entity sets.
 - ◆ E.g., (player) **number** and (team) **name** is a key for **Players** in the previous example.

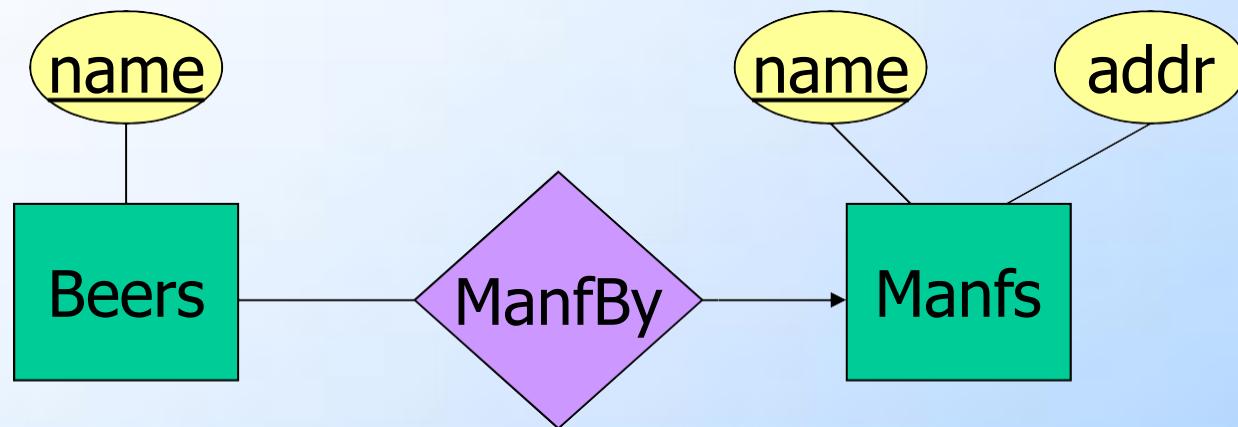
Design Techniques

1. Avoid redundancy.
2. Limit the use of weak entity sets.
3. Don't use an entity set when an attribute will do.

Avoiding Redundancy

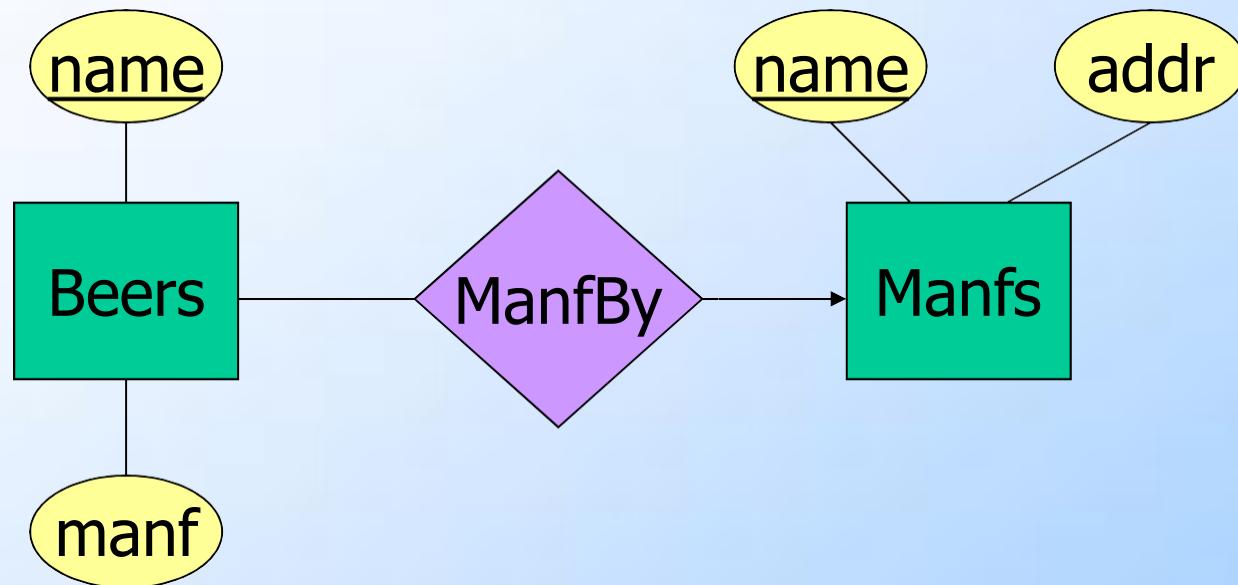
- ◆ *Redundancy* = saying the same thing in two (or more) different ways.
- ◆ Wastes space and (more importantly) encourages inconsistency.
 - ◆ Two representations of the same fact become inconsistent if we change one and forget to change the other.
 - ◆ Recall anomalies due to FD's.

Example: Good



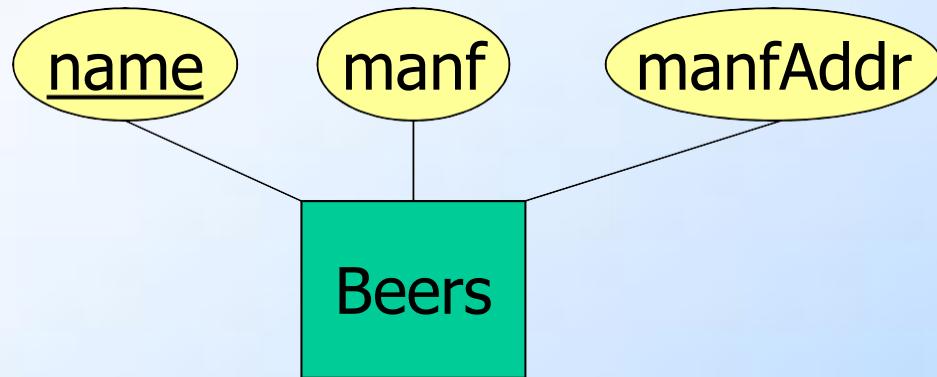
This design gives the address of each manufacturer exactly once.

Example: Bad



This design states the manufacturer of a beer twice: as an attribute and as a related entity.

Example: Bad



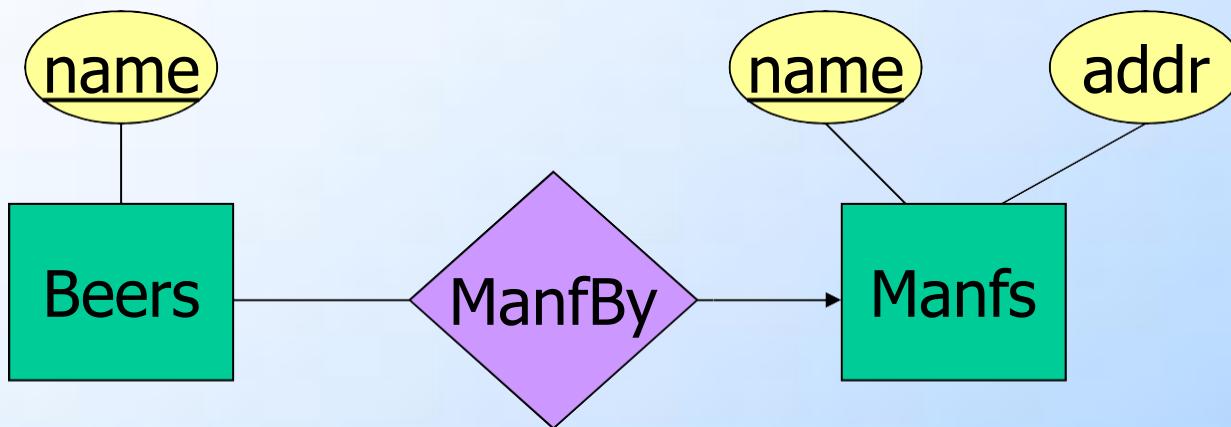
This design repeats the manufacturer's address once for each beer and loses the address if there are temporarily no beers for a manufacturer.

Entity Sets Versus Attributes

- ◆ An entity set should satisfy at least one of the following conditions:
 - ◆ It is more than the name of something; it has at least one nonkey attribute.

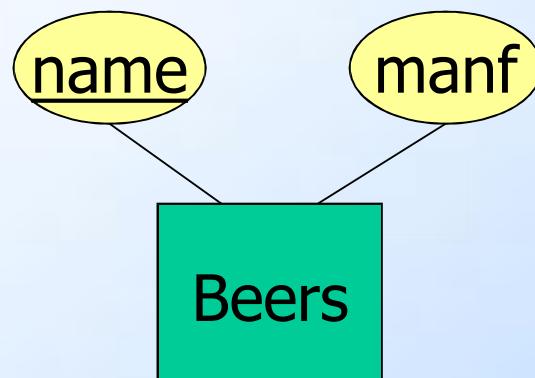
or
 - ◆ It is the “many” in a many-one or many-many relationship.

Example: Good



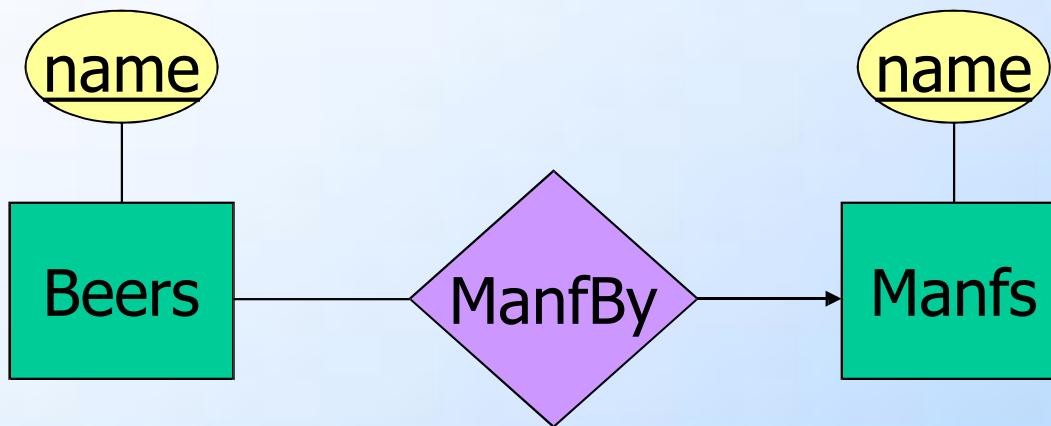
- **Manfs** deserves to be an entity set because of the nonkey attribute **addr**.
- **Beers** deserves to be an entity set because it is the “many” of the many-one relationship **ManfBy**.

Example: Good



There is no need to make the manufacturer an entity set, because we record nothing about manufacturers besides their name.

Example: Bad



Since the manufacturer is nothing but a name, and is not at the “many” end of any relationship, it should not be an entity set.

Don't Overuse Weak Entity Sets

- ◆ Beginning database designers often doubt that anything could be a key by itself.
 - ◆ They make all entity sets weak, supported by all other entity sets to which they are linked.
- ◆ In reality, we usually create unique ID's for entity sets.
 - ◆ Examples include social-security numbers, automobile VIN's etc.

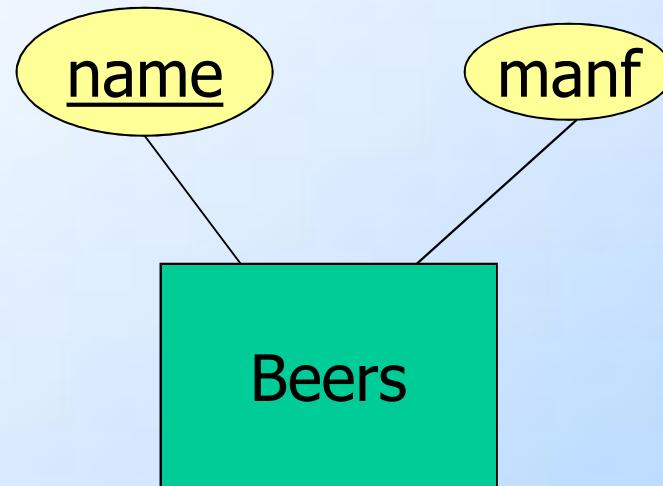
When Do We Need Weak Entity Sets?

- ◆ The usual reason is that there is no global authority capable of creating unique ID's.
- ◆ **Example:** it is unlikely that there could be an agreement to assign unique player numbers across all football teams in the world.

From E/R Diagrams to Relations

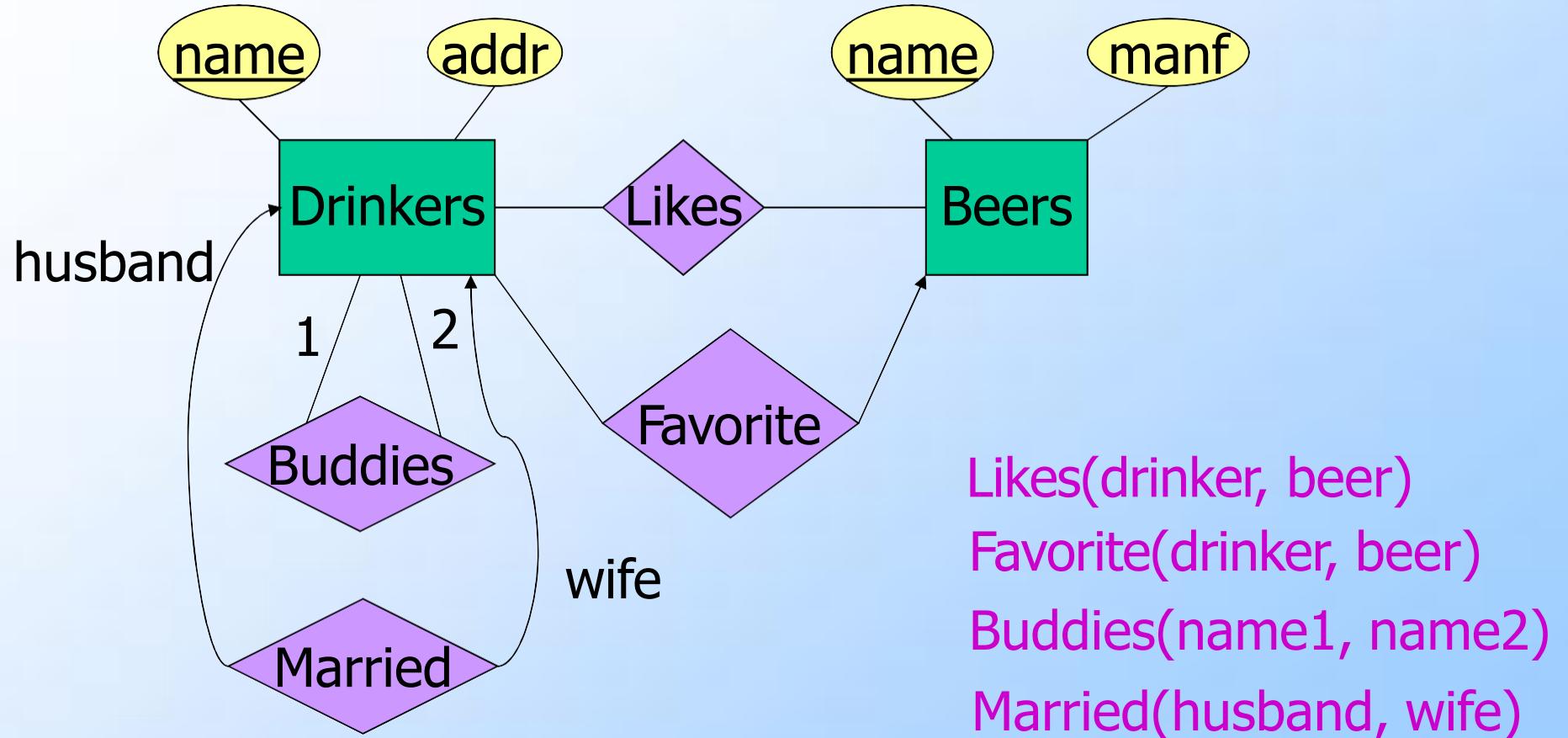
- ◆ Entity set -> relation.
 - ◆ Attributes -> attributes.
- ◆ Relationships -> relations whose attributes are only:
 - ◆ The keys of the connected entity sets.
 - ◆ Attributes of the relationship itself.

Entity Set -> Relation



Relation: Beers(name, manf)

Relationship -> Relation



Combining Relations

- ◆ OK to combine into one relation:
 1. The relation for an entity-set E
 2. The relations for many-one relationships of which E is the “many.”
- ◆ Example: $\text{Drinkers}(\text{name}, \text{addr})$ and $\text{Favorite}(\text{drinker}, \text{beer})$ combine to make $\text{Drinker1}(\text{name}, \text{addr}, \text{favBeer})$.

Risk with Many-Many Relationships

- ◆ Combining Drinkers with Likes would be a mistake. It leads to redundancy, as:

name	addr	beer
Sally	123 Maple	Bud
Sally	123 Maple	Miller

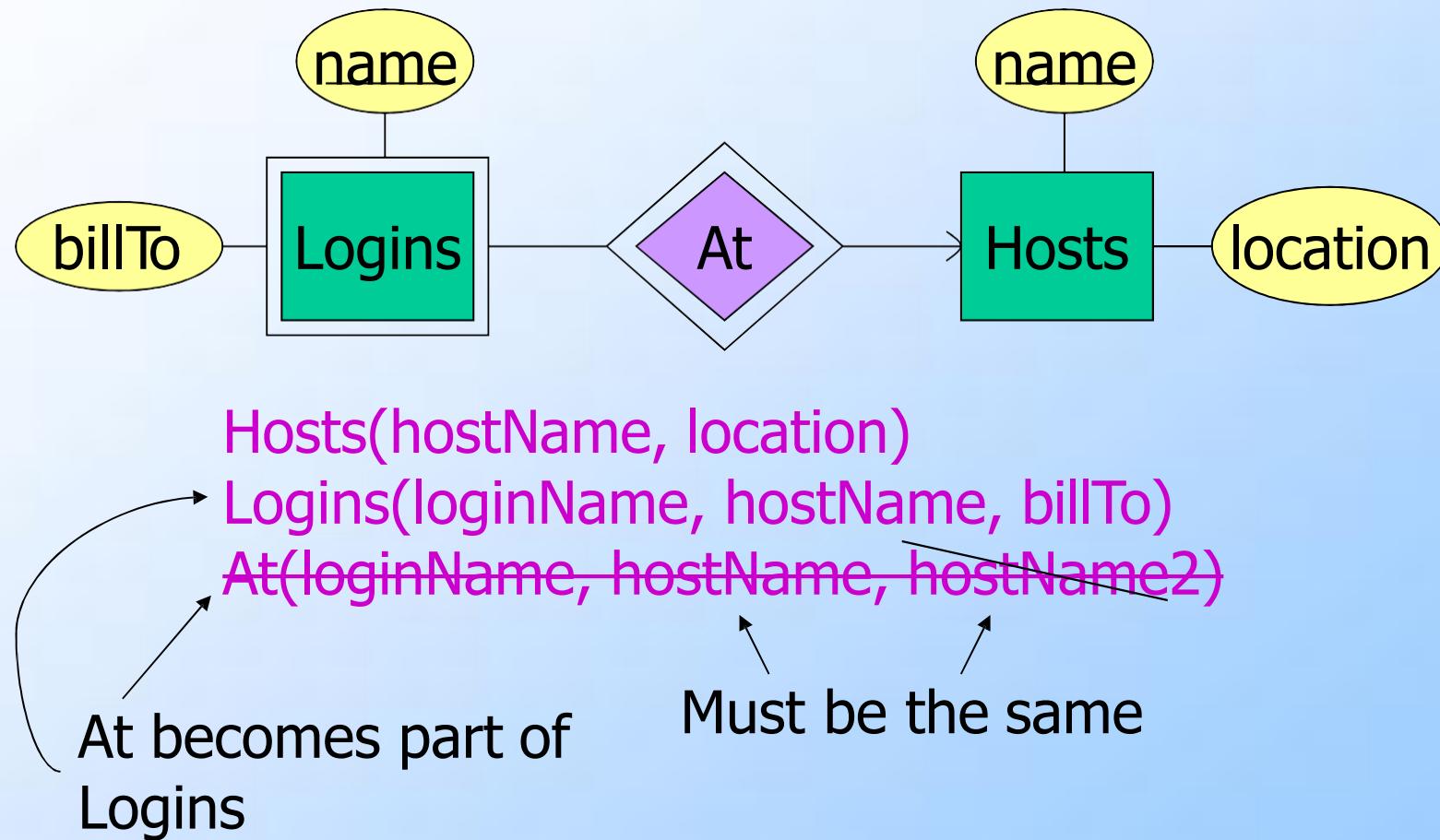
A yellow oval highlights the 'addr' column for both rows. An arrow points from the word 'Redundancy' at the bottom to this highlighted area.

Redundancy

Handling Weak Entity Sets

- ◆ Relation for a weak entity set must include attributes for its complete key (including those belonging to other entity sets), as well as its own, nonkey attributes.
- ◆ A supporting relationship is redundant and yields no relation (unless *it* has attributes).

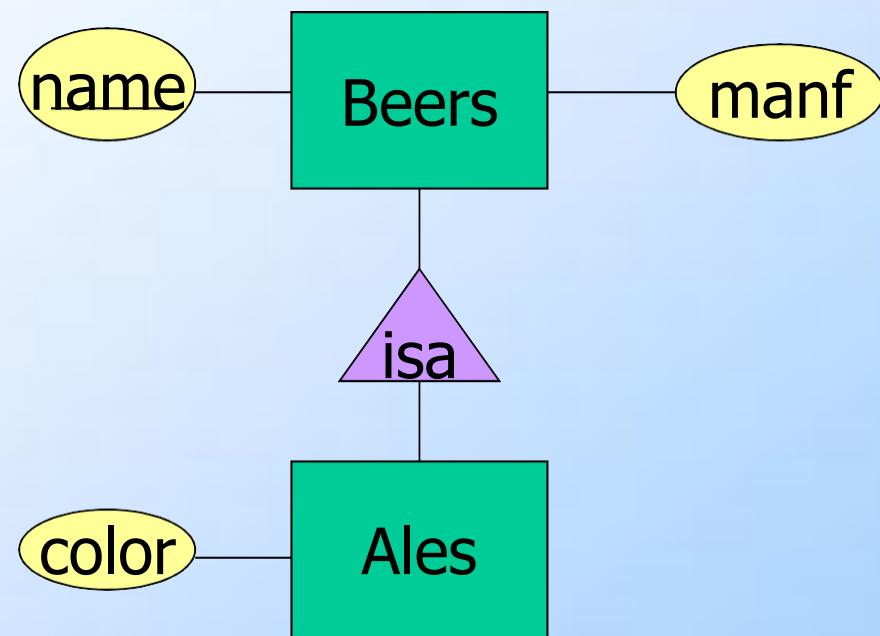
Example: Weak Entity Set -> Relation



Subclasses: Three Approaches

1. *Object-oriented*: One relation per subset of subclasses, with all relevant attributes.
2. *Use nulls*: One relation; entities have NULL in attributes that don't belong to them.
3. *E/R style*: One relation for each subclass:
 - ◆ Key attribute(s).
 - ◆ Attributes of that subclass.

Example: Subclass -> Relations



Object-Oriented

name	manf
Bud	Anheuser-Busch

Beers

name	manf	color
Summerbrew	Pete's	dark

Ales

Good for queries like “find the color of ales made by Pete’s.”

E/R Style

name	manf
Bud	Anheuser-Busch
Summerbrew	Pete's

Beers

name	color
Summerbrew	dark

Ales

Good for queries like
“find all beers (including
ales) made by Pete’s.”

Using Nulls

name	manf	color
Bud	Anheuser-Busch	NULL
Summerbrew	Pete's	dark

Beers

Saves space unless there are *lots* of attributes that are usually NULL.

Data models and Databases

Functional Dependencies

Relational Schema Design

- ▶ Goal of relational schema design is to avoid anomalies and redundancy.
 - ▶ *Update anomaly* : one occurrence of a fact is changed, but not all occurrences.
 - ▶ *Deletion anomaly* : valid fact is lost when a tuple is deleted.

Example of Bad Design

Drinkers(name, addr, beersLiked, manf, favBeer)

name	addr	beersLiked	manf	favBeer
Janeway	Voyager	Bud	A.B.	WickedAle
Janeway	???	WickedAle	Pete's	???
Spock	Enterprise	Bud	???	Bud

Data is redundant, because each of the ???'s can be figured out by using the FD's **name -> addr favBeer** and **beersLiked -> manf**.

This Bad Design Also Exhibits Anomalies

name	addr	beersLiked	manf	favBeer
Janeway	Voyager	Bud	A.B.	WickedAle
Janeway	Voyager	WickedAle	Pete's	WickedAle
Spock	Enterprise	Bud	A.B.	Bud

- **Update anomaly:** if Janeway is transferred to *Intrepid*, will we remember to change each of her tuples?
- **Deletion anomaly:** If nobody likes Bud, we lose track of the fact that Anheuser-Busch manufactures Bud.

Decomposition

- ▶ **Definition**

$d=\{R_1, \dots, R_k\}$ decomposition, if $R_1 \cup \dots \cup R_k = R$.

- ▶ **Example:**

$R=ABCDE$, $d=\{AD, BCE, ABE\}$

$R_1=AD$, $R_2=BCE$, $R_3=ABE$

Functional Dependencies

- ▶ $X \rightarrow Y$ is an assertion about a relation R that whenever two tuples of R agree on all the attributes of X , then they must also agree on all attributes in set Y .
- ▶ Say “ $X \rightarrow Y$ holds in R .”
- ▶ Convention: ..., X , Y , Z represent sets of attributes; A , B , C ,... represent single attributes.
- ▶ Convention: no set formers in sets of attributes, just ABC , rather than $\{A,B,C\}$.

Splitting Right Sides of FD's

- ▶ $X \rightarrow A_1 A_2 \dots A_n$ holds for R exactly when each of $X \rightarrow A_1$, $X \rightarrow A_2, \dots$, $X \rightarrow A_n$ hold for R .
- ▶ Example: $A \rightarrow BC$ is equivalent to $A \rightarrow B$ and $A \rightarrow C$.
- ▶ There is no splitting rule for left sides.
- ▶ We'll generally express FD's with singleton right sides.

Example: FD's

Drinkers(name, addr, beersLiked, manf, favBeer)

- ▶ Reasonable FD's to assert:
 1. $\text{name} \rightarrow \text{addr favBeer}$
 - ◆ Note this FD is the same as $\text{name} \rightarrow \text{addr}$ and $\text{name} \rightarrow \text{favBeer}$.
 2. $\text{beersLiked} \rightarrow \text{manf}$

Example: Possible Data

name	addr	beersLiked	manf	favBeer
Janeway	Voyager	Bud	A.B.	WickedAle
Janeway	Voyager	WickedAle	Pete's	WickedAle
Spock	Enterprise	Bud	A.B.	Bud

Because $\text{name} \rightarrow \text{addr}$

Because $\text{name} \rightarrow \text{favBeer}$

Because $\text{beersLiked} \rightarrow \text{manf}$

Keys of Relations

- ▶ K is a *superkey* for relation R if K functionally determines all of R .
- ▶ K is a *key* for R if K is a superkey, but no proper subset of K is a superkey.

Example: Superkey

Drinkers(name, addr, beersLiked, manf, favBeer)

- ▶ {name, beersLiked} is a superkey because together these attributes determine all the other attributes.
 - ▶ name \rightarrow addr favBeer
 - ▶ beersLiked \rightarrow manf

Example: Key

- ▶ $\{\text{name}, \text{beersLiked}\}$ is a **key** because neither $\{\text{name}\}$ nor $\{\text{beersLiked}\}$ is a superkey.
- ▶ name doesn't \rightarrow manf; beersLiked doesn't \rightarrow addr.
- ▶ There are no other keys, but lots of superkeys.
 - ▶ Any superset of $\{\text{name}, \text{beersLiked}\}$.

Where Do Keys Come From?

1. Just assert a key K .
 - ▶ The only FD's are $K \rightarrow A$ for all attributes A .
2. Assert FD's and deduce the keys by systematic exploration.

More FD's From “Physics”

- ▶ **Example:** “no two courses can meet in the same room at the same time” tells us:
hour room -> course.

- ▶ **ABC** relational schemas $AB \rightarrow C$ and $C \rightarrow B$
 - ▶ A = street, B = city, C = zip code.
- ▶ Keys: $\{A, B\}$ and $\{A, C\}$, too.

Inferring FD's

- ▶ We are given FD's
 $X_1 \rightarrow A_1, X_2 \rightarrow A_2, \dots, X_n \rightarrow A_n$,
and we want to know whether an FD $Y \rightarrow B$ must hold in any relation that satisfies the given FD's.
- ▶ Example: If $A \rightarrow B$ and $B \rightarrow C$ hold, surely $A \rightarrow C$ holds, even if we don't say so.
- ▶ Important for design of good relation schemas.

Armstrong axioms

Let $R(U)$ relation schema and $X, Y \subseteq U$, and denote
 XY is the union of attribute-sets X and Y

Let F functional dependencies

Armstrong axioms:

- ▶ A1 (reflexivity or trivial fd): if $Y \subseteq X$ then $X \rightarrow Y$.
- ▶ A2 (augmentation): if $X \rightarrow Y$ then $XZ \rightarrow YZ$.
- ▶ A3 (transitivity): if $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$.

More rules about functional dependencies

4. Splitting (decomposition) rule
 $X \rightarrow Y$ and $Z \subseteq Y$ then $X \rightarrow Z$.
5. Combining (union) rule
 $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$.
6. Pseudotranzititivity
 $X \rightarrow Y$ and $WY \rightarrow Z$ then $XW \rightarrow Z$.

Proof (4): Reflexivity axiom $Y \rightarrow Z$, and
tranzititivity axiom $X \rightarrow Z$.

Proof (5): Augmentation axioms $XX \rightarrow YX$ and $YX \rightarrow YZ$,
 $XX = X$, tranzititivity axioms $X \rightarrow YZ$.

Proof (6): Augmentation axioms $XW \rightarrow YW$ and $YW = WY$,
tranzititivity axiom $XW \rightarrow Z$.

Closure of set of attributes

- ▶ Definition: $X^+(F) := \{A \mid F \vdash X \rightarrow A\}$
- ▶ The *closure* of X under the FD's in S is the set of attributes A such that every relation that satisfies all the FD's in set S also satisfies $X \rightarrow A$, that is $X \rightarrow A$ follows from the FD's of S .

- ▶ Lemma: $F \vdash X \rightarrow Y \Leftrightarrow Y \subseteq X^+$.

Proof: (\Rightarrow) if $\forall A \in Y$ reflexivity and transitivity rule $F \vdash X \rightarrow A$, so $Y \subseteq X^+$.

(\Leftarrow) if $\forall A \in Y \subseteq X^+$ then $F \vdash X \rightarrow A$, union rule $F \vdash X \rightarrow Y$.

Closure Test for Y^+

- ▶ Input: Y set of attributes, F funct.dependencies
- ▶ Output: Y^+
- ▶ Algorithm Y^+ :

loop

$Y(0) := Y$

$Y(n+1) := Y(n) \cup \{A \mid \underline{X \rightarrow Z \in F}, A \in Z, X \subseteq Y(n)\}$

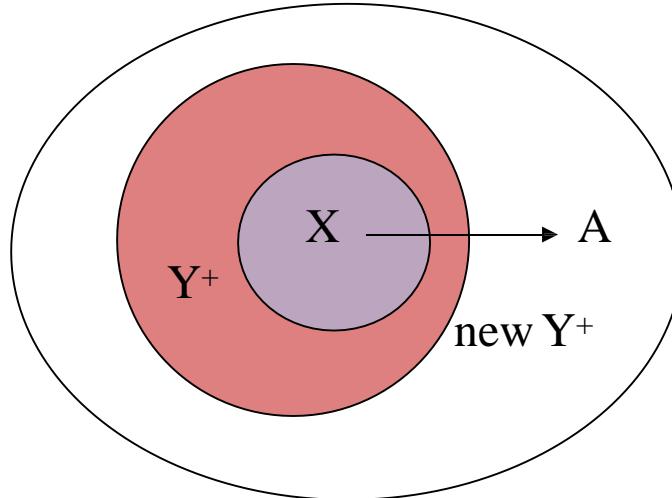
if $Y(v+1) = Y(v)$, then leave the loop

end of the loop

Output: $Y(v) = Y^+$.

Closure Test

- ▶ An easier way to test is to compute the *closure* of Y , denoted Y^+ .
- ▶ **Basis:** $Y^+ = Y$.
- ▶ **Induction:** Look for an FD's left side X that is a subset of the current Y^+ . If the FD is $X \rightarrow A$, add A to Y^+ .



Example: Closure Test

$R = ABCDEFG, \{AB \rightarrow C, B \rightarrow G, CD \rightarrow EG, BG \rightarrow E\}$

$X = ABF, X^+ = ?$

$X(0) := ABF$

$X(1) := ABF \cup \{C, G\} = ABCFG$

$X(2) := ABCFG \cup \{C, G, E\} = ABCEFG$

$X(3) := ABCEFG$

$X^+ = ABCEFG$

Exercises (Book 3.5.2.)

Consider the relation Courses(C, T, H, R, S, G),

$$F = \{C \rightarrow T, HR \rightarrow C, HT \rightarrow R, HS \rightarrow R, CS \rightarrow G\}$$

whose attributes may be thought of informally as course, teacher, hour, room, student, and grade.

Let the set of FD's for Courses be $C \rightarrow T, HR \rightarrow C, HT \rightarrow R, HS \rightarrow R$, and $CS \rightarrow G$.

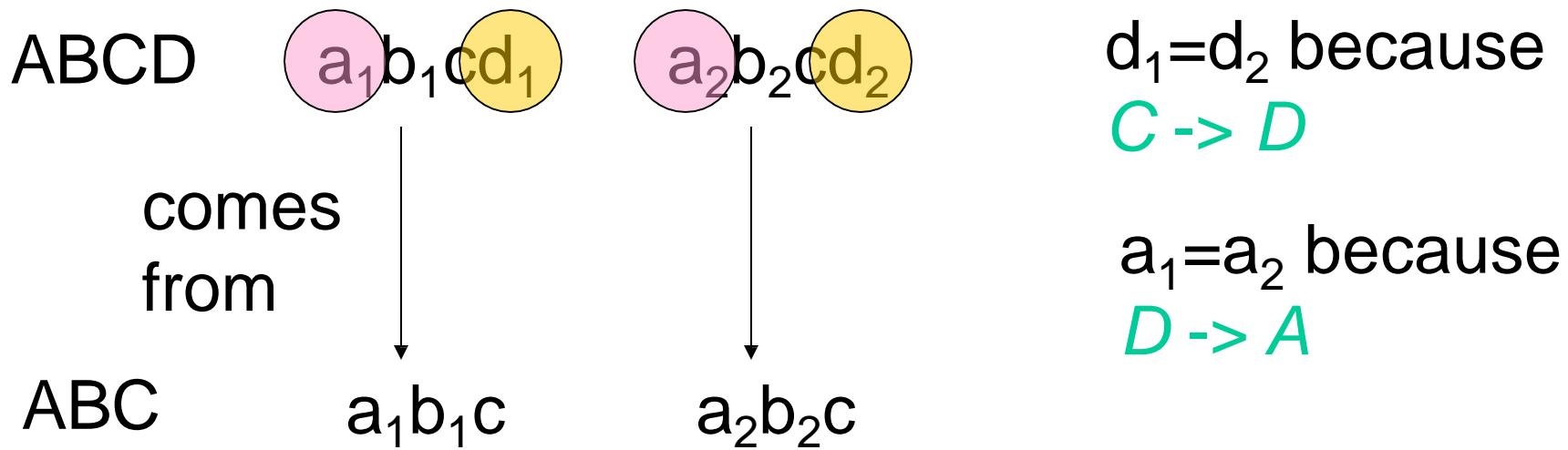
Intuitively, the first says that a course has a unique teacher, and the second says that only one course can meet in a given room at a given hour. The third says that a teacher can be in only one room at a given hour, and the fourth says the same about students. The last says that students get only one grade in a course.

What are all the keys for Courses?

Finding All Implied FD's - Projecting FD's...

- ▶ Motivation: “normalization,” the process where we break a relation schema into two or more schemas.
- ▶ Example: $ABCD$ with FD's $AB \rightarrow C$, $C \rightarrow D$, and $D \rightarrow A$.
 - ▶ Decompose into ABC , AD . What FD's hold in ABC ?
 - ▶ Not only $AB \rightarrow C$, but also $C \rightarrow A$!

Why?



Thus, tuples in the projection with equal C's have equal A's;
 $C \rightarrow A$.

Basic Idea

1. Start with given FD's and find all *nontrivial* FD's that follow from the given FD's.
 - ▶ Nontrivial = right side not contained in the left.
2. Restrict to those FD's that involve only attributes of the projected schema.

Simple, Exponential Algorithm

1. For each set of attributes X , compute X^+ .
2. Add $X \rightarrow A$ for all A in $X^+ - X$.
3. However, drop $XY \rightarrow A$ whenever we discover $X \rightarrow A$.
 - ◆ Because $XY \rightarrow A$ follows from $X \rightarrow A$ in any projection.
4. Finally, use only FD's involving projected attributes.

A Few Tricks

- ▶ No need to compute the closure of the empty set or of the set of all attributes.
- ▶ If we find $X^+ = \text{all attributes}$, so is the closure of any superset of X .

Example: Projecting FD's

- ▶ ABC with FD's $A \rightarrow B$ and $B \rightarrow C$. Project onto AC .
 - ▶ $A^+ = ABC$; yields $A \rightarrow B, A \rightarrow C$.
 - ▶ We do not need to compute AB^+ or AC^+ .
 - ▶ $B^+ = BC$; yields $B \rightarrow C$.
 - ▶ $C^+ = C$; yields nothing.
 - ▶ $BC^+ = BC$; yields nothing.
- ▶ Resulting FD's: $A \rightarrow B$, $A \rightarrow C$, and $B \rightarrow C$.
- ▶ Projection onto AC : $A \rightarrow C$.
 - ▶ Only FD that involves a subset of $\{A, C\}$.

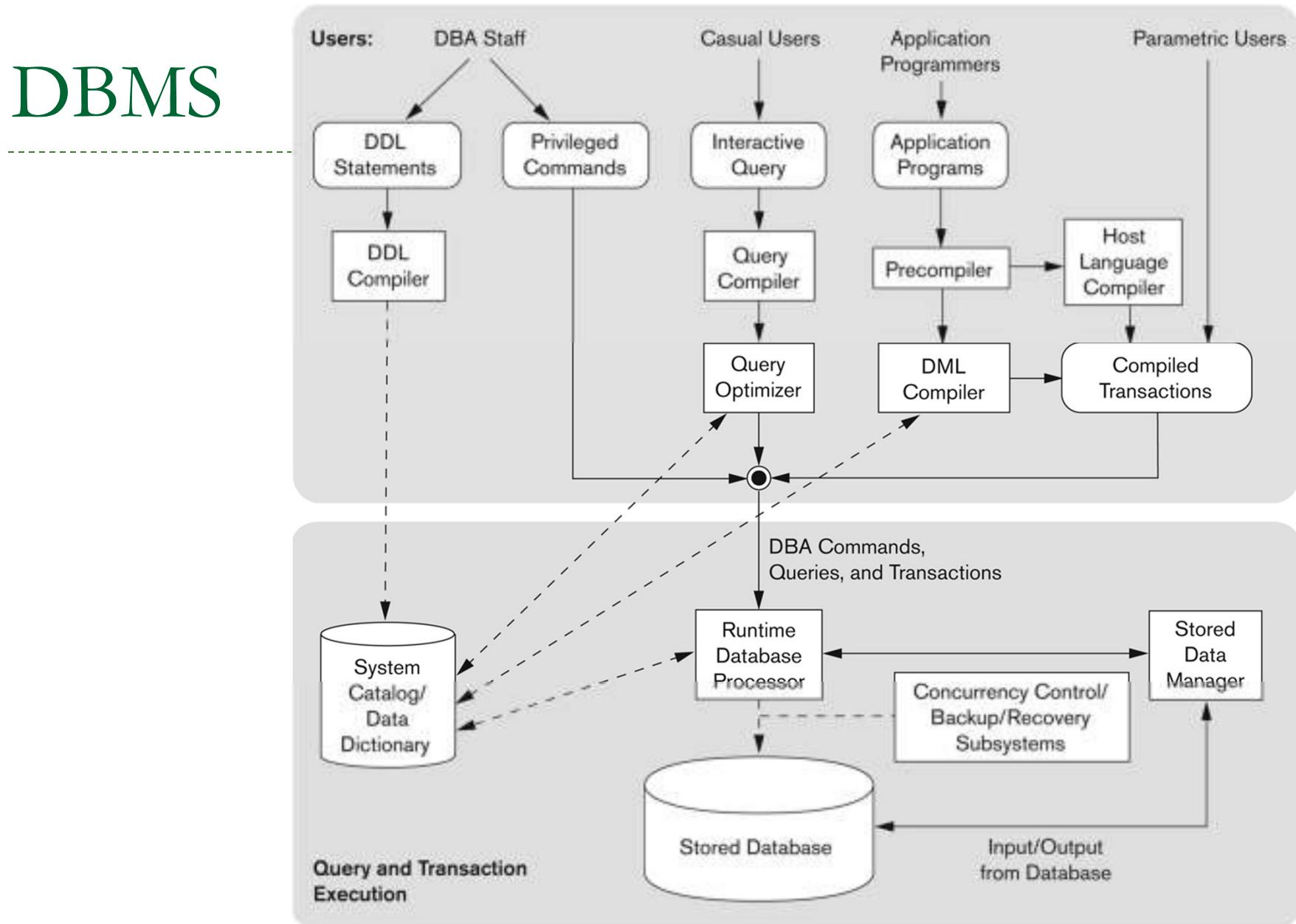


Data models and Databases



Defining Tables, Constraints

DBMS



► 2

Figure 2.3

Component modules of a DBMS and their interactions.

Rest of SQL

- ⌚ Defining a Database Schema
- ⌚ Primary Keys, Foreign Keys
- ⌚ Local and Global Constraints
- ⌚ Defining Views
- ⌚ Triggers

Defining a Database Schema

- ⌚ A database schema comprises declarations for the relations (“tables”) of the database.
- ⌚ Many other kinds of elements may also appear in the database schema, including views, constraints and triggers.

Declaring a Relation

⌚ Simplest form is:

⌚ CREATE TABLE <name> (<list of elements>);

```
CREATE TABLE Sells (
    bar        CHAR(20) ,
    beer       VARCHAR(20) ,
    price      REAL
) ;
```

Elements of Table Declarations

- ⌚ The principal element is a pair consisting of an attribute and a type.
- ⌚ The most common types are:
 - ⌚ INT or INTEGER (synonyms).
 - ⌚ REAL or FLOAT (synonyms).
 - ⌚ CHAR(n) = fixed-length string of n characters.
 - ⌚ VARCHAR(n) = variable-length string of up to n characters.

Dates and Times

- ⌚ DATE and TIME are types in SQL.
- ⌚ The form of a date value is:
DATE ‘yyyy-mm-dd’
- ⌚ Example: DATE ‘2002-09-30’ for Sept. 30, 2002.
- ⌚ There are functions to convert DATE types (e.g. TODATE)

Times as Values

- The form of a time value is:

TIME 'hh:mm:ss'

with an optional decimal point and fractions of a second following.

- Example: TIME '15:30:02.5' = two and a half seconds after 3:30PM.

Example: Create Table

```
CREATE TABLE Sells (  
    bar      CHAR(20) ,  
    beer     VARCHAR(20) ,  
    price    REAL  
) ;
```

Example: Create Table from existing table(s)

```
CREATE TABLE AVG_PRICES AS  
(SELECT beer, AVG(price)  
FROM Sells  
GROUP BY beer);
```

Remove a relation from schema

- ⌚ Remove a relation from the database schema by:
 - ⌚ `DROP TABLE <name>;`

- ⌚ Example:

```
DROP TABLE Sells;
```

Declaring Keys

- ☞ An attribute or list of attributes may be declared PRIMARY KEY or UNIQUE.
- ☞ These each say the attribute(s) so declared functionally determine all the attributes of the relation schema.
- ☞ There are a few distinctions to be mentioned later.

Declaring Single-Attribute Keys

- ☞ Place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute.
- ☞ Example:

```
CREATE TABLE Beers (
    name CHAR (20) UNIQUE,
    manf CHAR (20)
);
```

Declaring Multiatribute Keys

- ⌚ A key declaration can also be another element in the list of elements of a CREATE TABLE statement.
- ⌚ This form is essential if the key consists of more than one attribute.
- ⌚ May be used even for one-attribute keys.

Example: Multiatribute Key

- The bar and beer together are the key for Sells:

```
CREATE TABLE Sells (
    bar      CHAR(20),
    beer     VARCHAR(20),
    price    REAL,
    PRIMARY KEY (bar, beer)
) ;
```

PRIMARY KEY Versus UNIQUE

- ⌚ The SQL standard allows DBMS implementers to make their own distinctions between PRIMARY KEY and UNIQUE.
- ⌚ Example: some DBMS might automatically create an *index* (data structure to speed search) in response to PRIMARY KEY, but not UNIQUE.

Required Distinctions

- ☞ However, standard SQL requires these distinctions:
 1. There can be only one PRIMARY KEY for a relation, but several UNIQUE attributes.
 2. No attribute of a PRIMARY KEY can ever be NULL in any tuple. But attributes declared UNIQUE may have NULL's, and there may be several tuples with NULL.

Other Declarations for Attributes

- ⌚ Two other declarations we can make for an attribute are:
 1. NOT NULL means that the value for this attribute may never be NULL.
 2. DEFAULT <value> says that if there is no specific value known for this attribute's component in some tuple, use the stated <value>.

Example: Default Values

```
CREATE TABLE Drinkers (
    name CHAR(30) PRIMARY KEY,
    addr CHAR(50)
        DEFAULT '123 Sesame St.',
    phone CHAR(16)
) ;
```

Effect of Defaults -- 1

- ⌚ Suppose we insert the fact that Sally is a drinker, but we know neither her address nor her phone.
- ⌚ An INSERT with a partial list of attributes makes the insertion possible:

```
INSERT INTO Drinkers(name)  
VALUES ('Sally');
```

Effect of Defaults -- 2

- ⌚ But what tuple appears in Drinkers?

name	addr	phone
‘Sally’	‘123 Sesame St’	NULL

-
- ⌚ If we had declared phone NOT NULL, this insertion would have been rejected.

Adding Attributes

- We may change a relation schema by adding a new attribute (“column”) by:

```
ALTER TABLE <name> ADD  
    <attribute declaration>;
```

- Example:

```
ALTER TABLE Bars ADD  
    phone CHAR(16) DEFAULT 'unlisted';
```

Deleting/Renaming Attributes

- Remove an attribute from a relation schema by:

```
ALTER TABLE <name> DROP <attribute>;
```

- Example: we don't really need the license attribute for bars:

```
ALTER TABLE Bars DROP license;
```

- Rename an attribute in a relation:

```
ALTER TABLE <name> RENAME COLUMN <attribute>  
TO <attribute>;
```

- Example: we would rename the license attribute bar_license

```
ALTER TABLE Bars RENAME COLUMN license  
to bar_license;
```

Kinds of Constraints

A *constraint* is a relationship among data elements that the DBMS is required to enforce.

- ⌚ Key constraints, Foreign-key or referential-integrity.
- ⌚ Value-based constraints.
 - ⌚ Constrain values of a particular attribute.
- ⌚ Tuple-based constraints.
 - ⌚ Relationship among components.
 - ⌚ Easier to implement than many constraints.

Foreign Keys

- ☞ Consider Relation Sells(bar, beer, price).
- ☞ We might expect that a beer value is a real beer --- something appearing in Beers.name .
- ☞ A constraint that requires a beer in Sells to be a beer in Beers is called a *foreign -key* constraint.

Expressing Foreign Keys

- ⌚ Use the keyword REFERENCES, either:
 1. Within the declaration of an attribute, when only one attribute is involved.
 2. As an element of the schema, as:
FOREIGN KEY (<list of attributes>)
REFERENCES <relation> (<attributes>)
- ⌚ Referenced attributes must be declared PRIMARY KEY or UNIQUE.

Example: With Attribute

```
CREATE TABLE Beers (
    name      CHAR(20) PRIMARY KEY,
    manf      CHAR(20) );
CREATE TABLE Sells
(   bar      CHAR(20),
    beer     CHAR(20) REFERENCES Beers(name)
    price    REAL );
```

Example: As Element

```
CREATE TABLE Beers (
    name      CHAR(20) PRIMARY KEY,
    manf      CHAR(20) );
CREATE TABLE Sells
(   bar      CHAR(20),
    beer     CHAR(20),
    price    REAL,
    FOREIGN KEY(beer) REFERENCES
        Beers(name));
```

Enforcing Foreign-Key Constraints

- ☞ If there is a foreign-key constraint from attributes of relation R to the primary key of relation S , two violations are possible:
 1. An insert or update to R introduces values not found in S .
 2. A deletion or update to S causes some tuples of R to “dangle.”

Actions Taken -- 1

- ⌚ Suppose $R = \text{Sells}$, $S = \text{Beers}$.
- ⌚ An insert or update to Sells that introduces a nonexistent beer must be rejected.
- ⌚ A deletion or update to Beers that removes a beer value found in some tuples of Sells can be handled in three ways.

Actions Taken -- 2

- ⌚ The three possible ways to handle beers that suddenly cease to exist are:
 1. *Default* : Reject the modification.
 2. *Cascade* : Make the same changes in Sells.
 - ◆ Deleted beer: delete Sells tuple.
 - ◆ Updated beer: change value in Sells.
 3. *Set NULL* : Change the beer to NULL.

Example: Cascade

- ⌚ Suppose we delete the Bud tuple from Beers.
 - ⌚ Then delete all tuples from Sells that have beer = 'Bud'.
- ⌚ Suppose we update the Bud tuple by changing 'Bud' to 'Budweiser'.
 - ⌚ Then change all Sells tuples with beer = 'Bud' so that beer = 'Budweiser'.

Example: Set NULL

- ⌚ Suppose we delete the Bud tuple from Beers.
- ⌚ Change all tuples of Sells that have beer = 'Bud' to have beer = NULL.
- ⌚ Suppose we update the Bud tuple by changing 'Bud' to 'Budweiser'.
- ⌚ Same change.

Choosing a Policy

- ⌚ When we declare a foreign key, we may choose policies SET NULL or CASCADE independently for deletions and updates.
- ⌚ Follow the foreign-key declaration by:
`ON [UPDATE, DELETE][SET NULL, CASCADE]`
- ⌚ Two such clauses may be used.
- ⌚ Otherwise, the default (reject) is used.

Example

```
CREATE TABLE Sells (
    barCHAR(20),
    beer      CHAR(20),
    price     REAL,
    FOREIGN KEY(beer)
        REFERENCES Beers(name)
        ON DELETE SET NULL
        ON UPDATE CASCADE ) ;
```

Attribute-Based Checks

- ⌚ Put a constraint on the value of a particular attribute.
- ⌚ CHECK(<condition>) must be added to the declaration for the attribute.
- ⌚ The condition may use the name of the attribute, but any other relation or attribute name must be in a subquery.

Example

```
CREATE TABLE Sells (
    barCHAR(20),
    beer      CHAR(20) CHECK ( beer IN
        (SELECT name FROM Beers)) ,
    price     REAL CHECK ( price <= 5.00 )
) ;
```

Timing of Checks

- An attribute-based check is checked only when a value for that attribute is inserted or updated.
- Example: CHECK (price <= 5.00) checks every new price and rejects it if it is more than \$5.
- Example: CHECK (beer IN (SELECT name FROM Beers)) not checked if a beer is deleted from Beers (unlike foreign-keys).

Tuple-Based Checks

- CHECK (<condition>) may be added as another element of a schema definition.
- The condition may refer to any attribute of the relation, but any other attributes or relations require a subquery.
- Checked on insert or update only.

Example: Tuple-Based Check

- Only Joe's Bar can sell beer for more than \$5:

```
CREATE TABLE Sells (
    bar        CHAR(20),
    beer       CHAR(20),
    price      REAL,
    CHECK (bar = 'Joe''s Bar' OR
           price <= 5.00)
) ;
```

Assertions

- These are database-schema elements, like relations or views.

- Defined by:

```
CREATE ASSERTION <name>
```

```
    CHECK ( <condition> );
```

- Condition may refer to any relation or attribute in the database schema.

Example: Assertion

- In `Sells(bar, beer, price)`, no bar may charge an average of more than \$5.

```
CREATE ASSERTION NoRipoffBars CHECK (
    NOT EXISTS (
        SELECT bar FROM Sells
        GROUP BY bar
        HAVING 5.00 < AVG(price)
    ));

```

Bars with an average price above \$5

Example: Assertion

- In `Drinkers(name, addr, phone)` and `Bars(name, addr, license)`, there cannot be more bars than drinkers.

```
CREATE ASSERTION FewBar CHECK (
    (SELECT COUNT(*) FROM Bars) <=
    (SELECT COUNT(*) FROM Drinkers)
) ;
```

Timing of Assertion Checks

- ⌚ In principle, we must check every assertion after every modification to any relation of the database.
- ⌚ A clever system can observe that only certain changes could cause a given assertion to be violated.
- ⌚ Example: No change to Beers can affect FewBar. Neither can an insertion to Drinkers.

Views

- ⌚ A view is a “virtual table,” a relation that is defined in terms of the contents of other tables and views.
- ⌚ Declare by:
`CREATE VIEW <name> AS <query>;`
- ⌚ In contrast, a relation whose value is really stored in the database is called a *base table*.

Example: View Definition

- CanDrink(drinker, beer) is a view “containing” the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS  
    SELECT drinker, beer  
    FROM Frequent, Sells  
    WHERE Frequent.bar = Sells.bar;
```

Example: Accessing a View

- ⌚ You may query a view as if it were a base table.
- ⌚ There is a limited ability to modify views if the modification makes sense as a modification of the underlying base table.
- ⌚ Example:

```
SELECT beer FROM CanDrink  
WHERE drinker = 'Sally';
```

What Happens When a View Is Used?

- The DBMS starts by interpreting the query as if the view were a base table.
- Typical DBMS turns the query into something like relational algebra.
- The queries defining any views used by the query are also replaced by their algebraic equivalents, and “spliced into” the expression tree for the query.

Constraints and Triggers

- ⌚ A *constraint* is a relationship among data elements that the DBMS is required to enforce.
 - ⌚ Example: key constraints.
- ⌚ *Triggers* are only executed when a specified condition occurs, e.g., insertion of a tuple.
 - ⌚ Easier to implement than many constraints.

Triggers: Motivation

- ⌚ Attribute- and tuple-based checks have limited capabilities.
- ⌚ Assertions are sufficiently general for most constraint applications, but they are hard to implement efficiently.
- ⌚ The DBMS must have real intelligence to avoid checking assertions that couldn't possibly have been violated.

Triggers: Motivation

- ⌚ Attribute- and tuple-based checks have limited capabilities.
- ⌚ Assertions are sufficiently general for most constraint applications, but they are hard to implement efficiently.
- ⌚ The DBMS must have real intelligence to avoid checking assertions that couldn't possibly have been violated.

Triggers: Solution

- A trigger allows the user to specify when the check occurs.
- Like an assertion, a trigger has a general-purpose condition and can also perform any sequence of SQL database modifications.

Event-Condition-Action Rules

- ⌚ Another name for “trigger” is *ECA rule*, or event-condition-action rule.
- ⌚ *Event*: typically a type of database modification, e.g., “insert on Sells.”
- ⌚ *Condition* : Any SQL boolean-valued expression.
- ⌚ *Action* : Any SQL statements.

Example: A Trigger

- There are many details to learn about triggers.
- Here is an example to set the stage.
- Instead of using a foreign-key constraint and rejecting insertions into `Sells(bar, beer, price)` with unknown beers, a trigger can add that beer to `Beers`, with a NULL manufacturer.

Example: Trigger Definition

```
CREATE TRIGGER BeerTrig  
    AFTER INSERT ON Sells  
    REFERENCING NEW ROW AS NewTuple  
    FOR EACH ROW  
        WHEN (NewTuple.beer NOT IN  
              (SELECT name FROM Beers))  
        INSERT INTO Beers(name)  
                  VALUES(NewTuple.beer);
```

The event

The condition

The action

Options: CREATE TRIGGER

- **CREATE TRIGGER <name>**
- Option:
 CREATE OR REPLACE TRIGGER <name>
- Useful if there is a trigger with that name and
 you want to modify the trigger.

Options: The Condition

- ⌚ AFTER can be BEFORE.
 - ⌚ Also, INSTEAD OF, if the relation is a view.
 - ⌚ A great way to execute view modifications: have triggers translate them to appropriate modifications on the base tables.
- ⌚ INSERT can be DELETE or UPDATE.
 - ⌚ And UPDATE can be UPDATE . . . ON a particular attribute.

Options: FOR EACH ROW

- ⌚ Triggers are either *row-level* or *statement-level*.
- ⌚ FOR EACH ROW indicates row-level; its absence indicates statement-level.
- ⌚ Row level triggers are executed once for each modified tuple.
- ⌚ Statement-level triggers execute once for an SQL statement, regardless of how many tuples are modified.

Options: REFERENCING

- ⌚ INSERT statements imply a new tuple (for row-level) or new set of tuples (for statement-level).
- ⌚ DELETE implies an old tuple or table.
- ⌚ UPDATE implies both.
- ⌚ Refer to these by
[NEW OLD][TUPLE TABLE] AS <name>

Options: The Condition

- ⌚ Any boolean-valued condition is appropriate.
- ⌚ It is evaluated before or after the triggering event, depending on whether BEFORE or AFTER is used in the event.
- ⌚ Access the new/old tuple or set of tuples through the names declared in the REFERENCING clause.

Options: The Action

- ⌚ There can be more than one SQL statement in the action.
- ⌚ Surround by BEGIN . . . END if there is more than one.
- ⌚ But queries make no sense in an action, so we are really limited to modifications.

Another Example

- Using Sells(bar, beer, price) and a unary relation RipoffBars(bar) created for the purpose, maintain a list of bars that raise the price of any beer by more than \$1.

The Trigger

CREATE TRIGGER PriceTrig

AFTER UPDATE OF price ON Sells

REFERENCING

OLD ROW as old

NEW ROW as new

FOR EACH ROW

WHEN(new.price > old.price + 1.00)

INSERT INTO RipoffBars

VALUES(new.bar);

The event –
only changes
to prices

Updates let us
talk about old
and new tuples

We need to consider
each price change

Condition:
a raise in
price > \$1

When the price change
is great enough, add
the bar to RipoffBars

Triggers on Views

- ⌚ Generally, it is impossible to modify a view, because it doesn't exist.
- ⌚ But an INSTEAD OF trigger lets us interpret view modifications in a way that makes sense.
- ⌚ Example: We'll design a view Synergy that has (drinker, beer, bar) triples such that the bar serves the beer, the drinker frequents the bar and likes the beer.

Example: The View

```
CREATE VIEW Synergy AS  
  SELECT Likes.drinker, Likes.beer, Sells.bar  
    FROM Likes, Sells, Frequents  
   WHERE Likes.drinker = Frequents.drinker  
         AND Likes.beer = Sells.beer  
         AND Sells.bar = Frequents.bar;
```

Pick one copy of each attribute

Natural join of Likes,
Sells, and Frequents

Interpreting a View Insertion

- ⌚ We cannot insert into Synergy --- it is a view.
- ⌚ But we can use an INSTEAD OF trigger to turn a (drinker, beer, bar) triple into three insertions of projected pairs, one for each of Likes, Sells, and Frequent.
- ⌚ The Sells.price will have to be NULL.

The Trigger

```
CREATE TRIGGER ViewTrig
  INSTEAD OF INSERT ON Synergy
  REFERENCING NEW ROW AS n
  FOR EACH ROW
  BEGIN
    INSERT INTO LIKES VALUES(n.drinker, n.beer);
    INSERT INTO SELLS(bar, beer) VALUES(n.bar, n.beer);
    INSERT INTO FREQUENTS VALUES(n.drinker, n.bar);
  END;
```

Data models and Databases

Relational Schema Design

Boyce-Codd Normal Form

- ▶ We say a relation R is in ***BCNF*** if whenever $X \rightarrow Y$ is a nontrivial FD that holds in R , X is a superkey.
- ▶ Remember: *nontrivial* means Y is not contained in X .
- ▶ Remember, a *superkey* is any superset of a key (not necessarily a proper superset).

Example

Drinkers(name, addr, beersLiked, manf, favBeer)

FD's: name->addr favBeer, beersLiked->manf

- ▶ Only key is {name, beersLiked}.
- ▶ In each FD, the left side is *not* a superkey.
- ▶ Any one of these FD's shows *Drinkers* is not in BCNF

Another Example

Beers(name, manf, manfAddr)

FD's: name->manf, manf->manfAddr

- ▶ Only key is {name} .
- ▶ name->manf does not violate BCNF, but manf->manfAddr does.

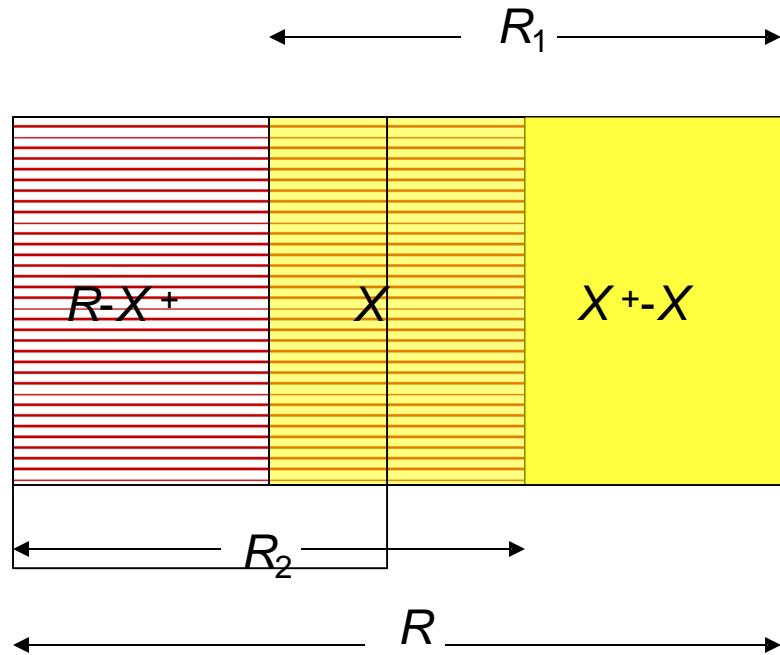
Decomposition into BCNF

- ▶ Given: relation R with FD's F .
- ▶ Look among the given FD's for a BCNF violation $X \rightarrow Y$.
 - ▶ If any FD following from F violates BCNF, then there will surely be an FD in F itself that violates BCNF.
- ▶ Compute X^+ .
 - ▶ Not all attributes, or else X is a superkey.

Decompose R Using $X \rightarrow Y$

- ▶ Replace R by relations with schemas:
 1. $R_1 = X^+$.
 2. $R_2 = R - (X^+ - X)$.
- ▶ *Project* given FD's F onto the two new relations.

Decomposition Picture



Example: BCNF Decomposition

Drinkers(name, addr, beersLiked, manf, favBeer)

$F = \text{name} \rightarrow \text{addr}, \quad \text{name} \rightarrow \text{favBeer},$
 $\text{beersLiked} \rightarrow \text{manf}$

- ▶ Pick BCNF violation $\text{name} \rightarrow \text{addr}$.
- ▶ Close the left side: $\{\text{name}\}^+ = \{\text{name}, \text{addr}, \text{favBeer}\}$.
- ▶ Decomposed relations:
 1. Drinkers1(name, addr, favBeer)
 2. Drinkers2(name, beersLiked, manf)

Example --- Continued

- ▶ We are not done; we need to check Drinkers1 and Drinkers2 for BCNF.
- ▶ Projecting FD's is easy here.
- ▶ For $\text{Drinkers1}(\underline{\text{name}}, \text{addr}, \text{favBeer})$, relevant FD's are $\text{name} \rightarrow \text{addr}$ and $\text{name} \rightarrow \text{favBeer}$.
 - ▶ Thus, $\{\text{name}\}$ is the only key and Drinkers1 is in BCNF.

Example --- Continued

- ▶ For $\text{Drinkers2}(\underline{\text{name}}, \underline{\text{beersLiked}}, \text{manf})$, the only FD is $\text{beersLiked} \rightarrow \text{manf}$, and the only key is $\{\text{name}, \text{beersLiked}\}$.
 - ▶ Violation of BCNF.
- ▶ $\text{beersLiked}^+ = \{\text{beersLiked}, \text{manf}\}$, so we decompose Drinkers2 into:
 1. $\text{Drinkers3}(\underline{\text{beersLiked}}, \text{manf})$
 2. $\text{Drinkers4}(\underline{\text{name}}, \underline{\text{beersLiked}})$

Example --- Concluded

- ▶ The resulting decomposition of *Drinkers* :
 1. *Drinkers1(name, addr, favBeer)*
 2. *Drinkers3(beersLiked, manf)*
 3. *Drinkers4(name, beersLiked)*
- ▶ Notice: *Drinkers1* tells us about drinkers, *Drinkers3* tells us about beers, and *Drinkers4* tells us the relationship between drinkers and the beers they like.

Testing for a Lossless Join

- ▶ If we project R onto R_1, R_2, \dots, R_k , can we recover R by rejoining?
- ▶ Any tuple in R can be recovered from its projected fragments.
- ▶ So the only question is: when we rejoin, do we ever get back something we didn't have originally?

The Chase Test

- ▶ Suppose tuple t comes back in the join.
- ▶ Then t is the join of projections of some tuples of R , one for each R_i of the decomposition.
- ▶ Can we use the given FD's to show that one of these tuples must be t ?

The Chase – (2)

- ▶ Start by assuming $t = abc\dots$.
- ▶ For each i , there is a tuple s_i of R that has a, b, c, \dots in the attributes of R_i .
- ▶ s_i can have any values in other attributes.
- ▶ We'll use the same letter as in t , but with a subscript, for these components.

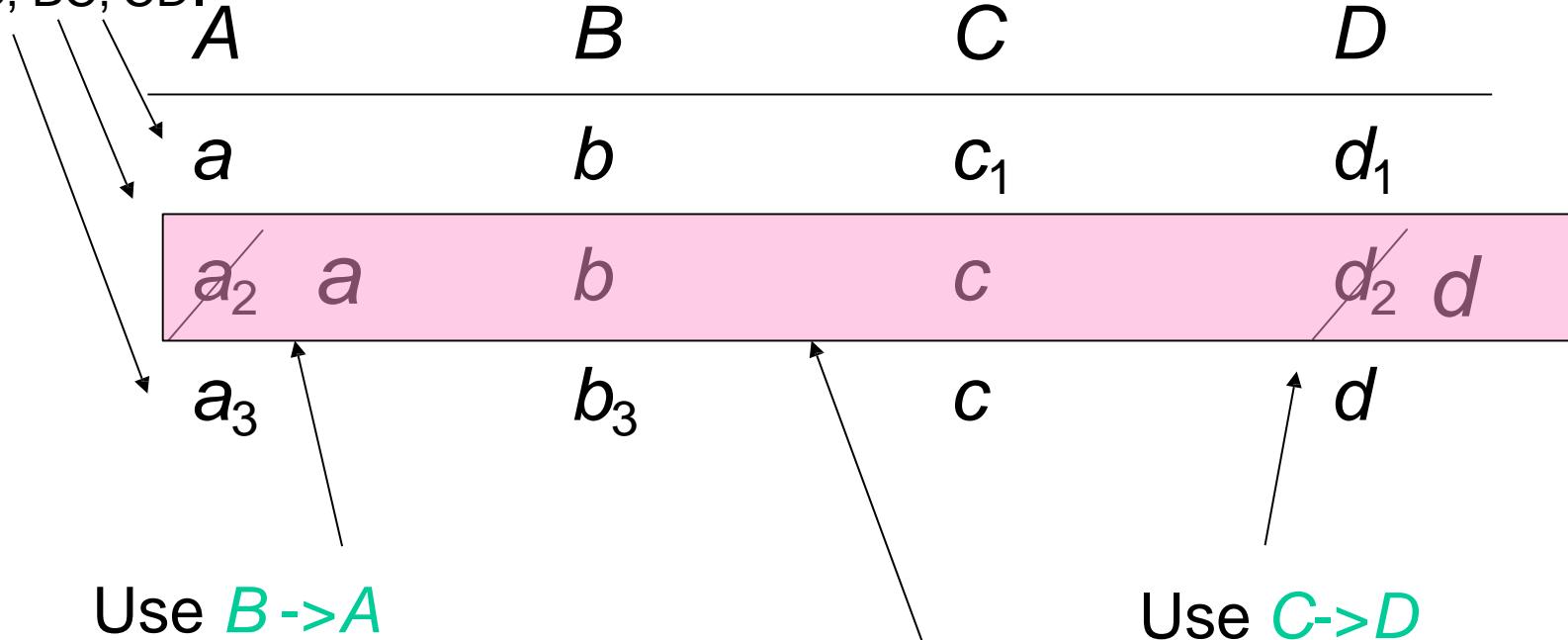
Example: The Chase

- ▶ Let $R = ABCD$, and the decomposition be AB , BC , and CD .
- ▶ Let the given FD's be $C \rightarrow D$ and $B \rightarrow A$.
- ▶ Suppose the tuple $t = abcd$ is the join of tuples projected onto AB , BC , CD .

The tuples
of R pro-
jected onto

The *Tableau*

AB, BC, CD.



We've proved the
second tuple must be t .

Summary of the Chase

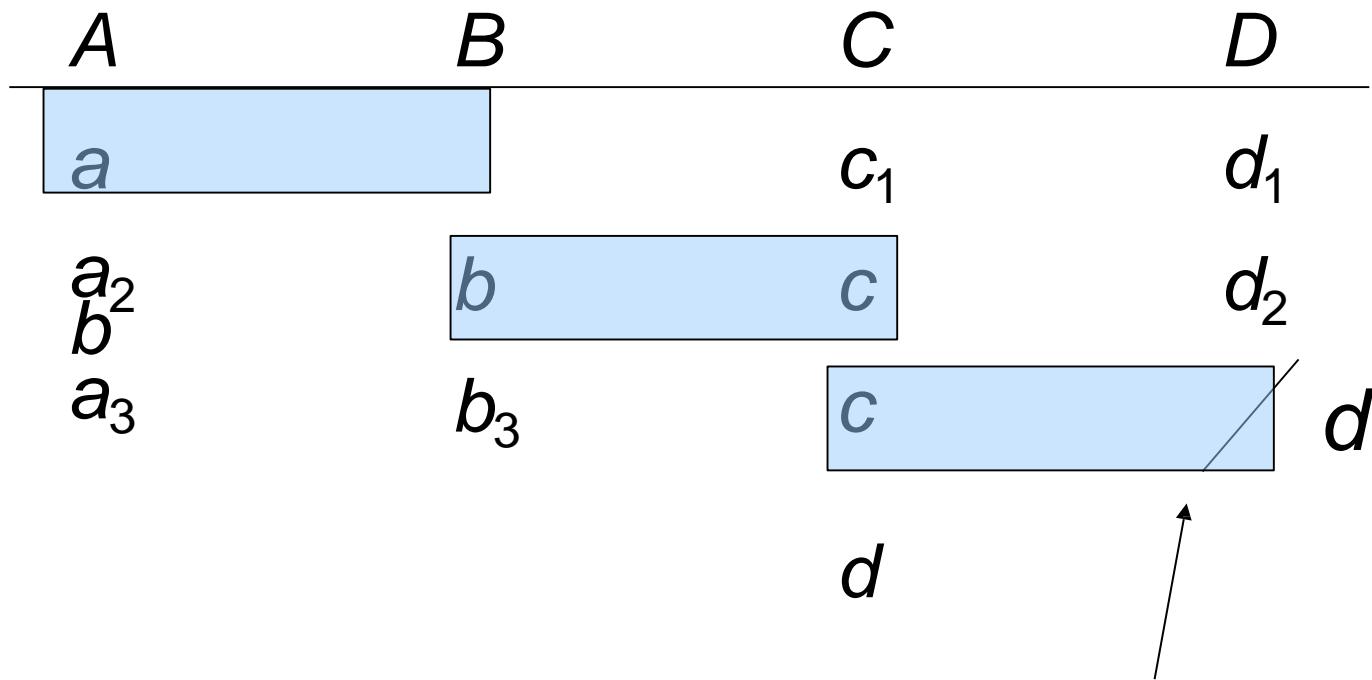
1. If two rows agree in the left side of a FD, make their right sides agree too.
2. Always replace a subscripted symbol by the corresponding unsubscripted one, if possible.
3. If we ever get an unsubscripted row, we know any tuple in the project-join is in the original (the join is lossless).
4. Otherwise, the final tableau is a counterexample.

Example: Lossy Join

- ▶ Same relation $R = ABCD$ and same decomposition.
- ▶ But with only the FD $C \rightarrow D$.

These projections
rejoin to form
 $abcd$.

The *Tableau*



These three tuples are an example
 R that shows the join lossy. $abcd$
is not in R , but we can project and
rejoin to get $abcd$.

Use $C > D$

Third Normal Form -- Motivation

- ▶ There is one structure of FD's that causes trouble when we decompose.
- ▶ $AB \rightarrow C$ and $C \rightarrow B$.
 - ▶ Example: A = street address, B = city, C = zip code.
- ▶ There are two keys, $\{A,B\}$ and $\{A,C\}$.
- ▶ $C \rightarrow B$ is a BCNF violation, so we must decompose into AC , BC .

We Cannot Enforce FD's

- ▶ The problem is that if we use AC and BC as our database schema, we cannot enforce the FD $AB \rightarrow C$ by checking FD's in these decomposed relations.
- ▶ Example with A = street, B = city, and C = zip on the next slide.

An Unenforceable FD

street	zip
545 Tech Sq	.
545 Tech Sq	02138

city	zip
Cambridge	02138
Cambridge	02139

Join tuples with equal zip codes.

street	city	zip
545 Tech Sq.	Cambridge	02138
545 Tech Sq.	Cambridge	02139

Although no FD's were violated in the decomposed relations,
FD **street city -> zip** is violated by the database as a whole.

3NF Let's Us Avoid This Problem

- ▶ 3rd Normal Form (3NF) modifies the BCNF condition so we do not have to decompose in this problem situation.
- ▶ An attribute is *prime* if it is a member of any key.
- ▶ $X \rightarrow A$ violates 3NF if and only if X is not a superkey, and also A is not prime.

Example: 3NF

- ▶ In our problem situation with FD's $AB \rightarrow C$ and $C \rightarrow B$, we have keys AB and AC .
- ▶ Thus A , B , and C are each prime.
- ▶ Although $C \rightarrow B$ violates BCNF, it does not violate 3NF.

What 3NF and BCNF Give You

- ▶ There are two important properties of a decomposition:
 1. *Lossless Join* : it should be possible to project the original relations onto the decomposed schema, and then reconstruct the original.
 2. *Dependency Preservation* : it should be possible to check in the projected relations whether all the given FD's are satisfied.

3NF and BCNF -- Continued

- ▶ We can get (1) with a BCNF decomposition.
- ▶ We can get both (1) and (2) with a 3NF decomposition.
- ▶ But we can't always get (1) and (2) with a BCNF decomposition.
 - ▶ street-city-zip is an example.

3NF Synthesis Algorithm

- ▶ We can always construct a decomposition into 3NF relations with a lossless join and dependency preservation.
- ▶ Need *minimal basis* for the FD's:
 1. Right sides are single attributes.
 2. No FD can be removed.
 3. No attribute can be removed from a left side.

Constructing a Minimal Basis

1. Split right sides.
2. Repeatedly try to remove an FD and see if the remaining FD's are equivalent to the original.
3. Repeatedly try to remove an attribute from a left side and see if the resulting FD's are equivalent to the original.

3NF Synthesis – (2)

- ▶ One relation for each FD in the minimal basis.
 - ▶ Schema is the union of the left and right sides.
- ▶ If no key is contained in an FD, then add one relation whose schema is some key.

Example: 3NF Synthesis

- ▶ Relation R = ABCD.
- ▶ FD's $A \rightarrow B$ and $A \rightarrow C$.
- ▶ Decomposition: AB and AC from the FD's, plus AD for a key.

Why It Works

- ▶ **Preserves dependencies:** each FD from a minimal basis is contained in a relation, thus preserved.
- ▶ **Lossless Join:** use the chase to show that the row for the relation that contains a key can be made all-unsubscripted variables.
- ▶ **3NF:** hard part – a property of minimal bases.

Indexing

An Introduction

Peter Lehotay-Kery

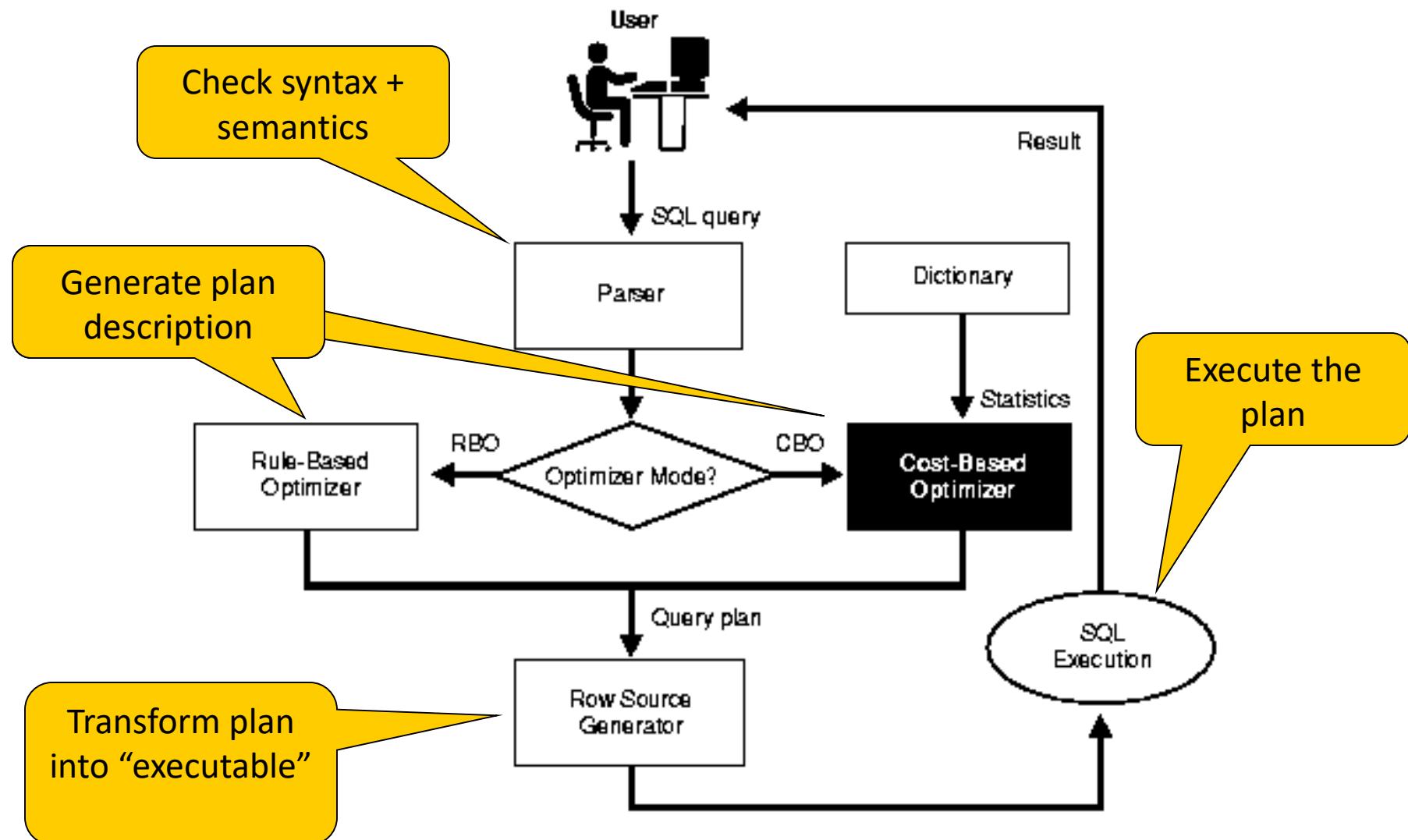
Based on the slides of

Toon Koppelaars

Sr. IT Architect

Central Bookhouse

Optimizer Overview



Cost vs. Rule

- Rule
 - Hardcoded heuristic rules determine plan
 - “Access via index is better than full table scan”
 - “Fully matched index is better than partially matched index”
 - ...
- Cost (2 modes)
 - Statistics of data play role in plan determination
 - Best throughput mode: retrieve **all rows** asap
 - First compute, then return fast
 - Best response mode: retrieve **first row** asap
 - Start returning while computing (if possible)

How to set which one?

- Instance level: Optimizer_Mode parameter
 - Rule
 - Choose
 - if statistics then CBO (all_rows), else RBO
 - First_rows, First_rows_n (1, 10, 100, 1000)
 - All_rows
- Session level:
 - Alter session set optimizer_mode=<mode>;
- Statement level:
 - Hints inside SQL text specify mode to be used

DML vs. Queries

- Open => Parse => Execute (=> Fetchⁿ)

```
SELECT ename, salary  
FROM emp  
WHERE salary>100000
```

Fetches done
By client

```
UPDATE emp  
SET commission='N'  
WHERE salary>100000
```

Same SQL optimization

All fetches done internally
by SQL-Executor

CLIENT

=> SQL =>
<= Data or Returncode<=

SERVER

Data Storage: Tables

- Oracle stores all data inside datafiles
 - Location & size determined by DBA
 - Logically grouped in tablespaces
 - Each file is identified by a relative file number (fno)
- Datafile consists of data-blocks
 - Size equals value of *db_block_size* parameter
 - Each block is identified by its offset in the file
- Data-blocks contain rows
 - Each row is identified by its sequence in the block

ROWID: <Block>.<Row>.<File>

Data Storage: Tables

File x

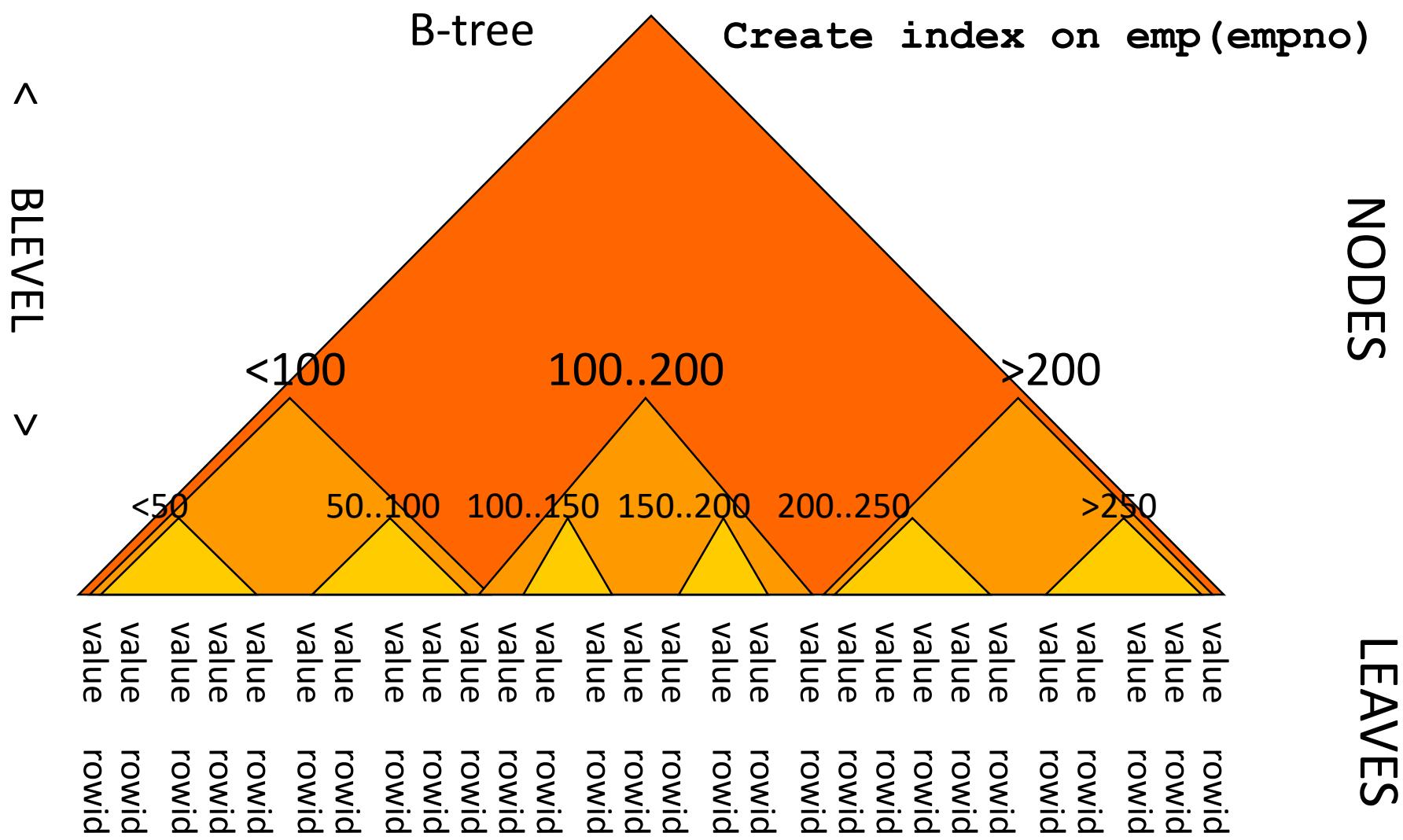
Block 1	Block 2	Block 3	Block 4
Block 5	Block ...	<Rec1><Rec2><Rec3> <Rec4><Rec5><Rec6> <Rec7><Rec8><Rec9> ...	

Rowid: 00000006.0000.000X

Data Storage: Indexes

- Balanced trees
 - Indexed column(s) sorted and stored separately
 - NULL values are excluded (not added to the index)
 - Pointer structure enables logarithmic search
 - Access index first, find pointer to table, then access table
- B-trees consist of
 - Node blocks
 - Contain pointers to other node, or leaf blocks
 - Leaf blocks
 - Contain actual indexed values
 - Contain rowids (pointer to rows)
- Also stored in blocks in datafiles
 - Proprietary format

Data Storage: Indexes



Data Storage: Indexes

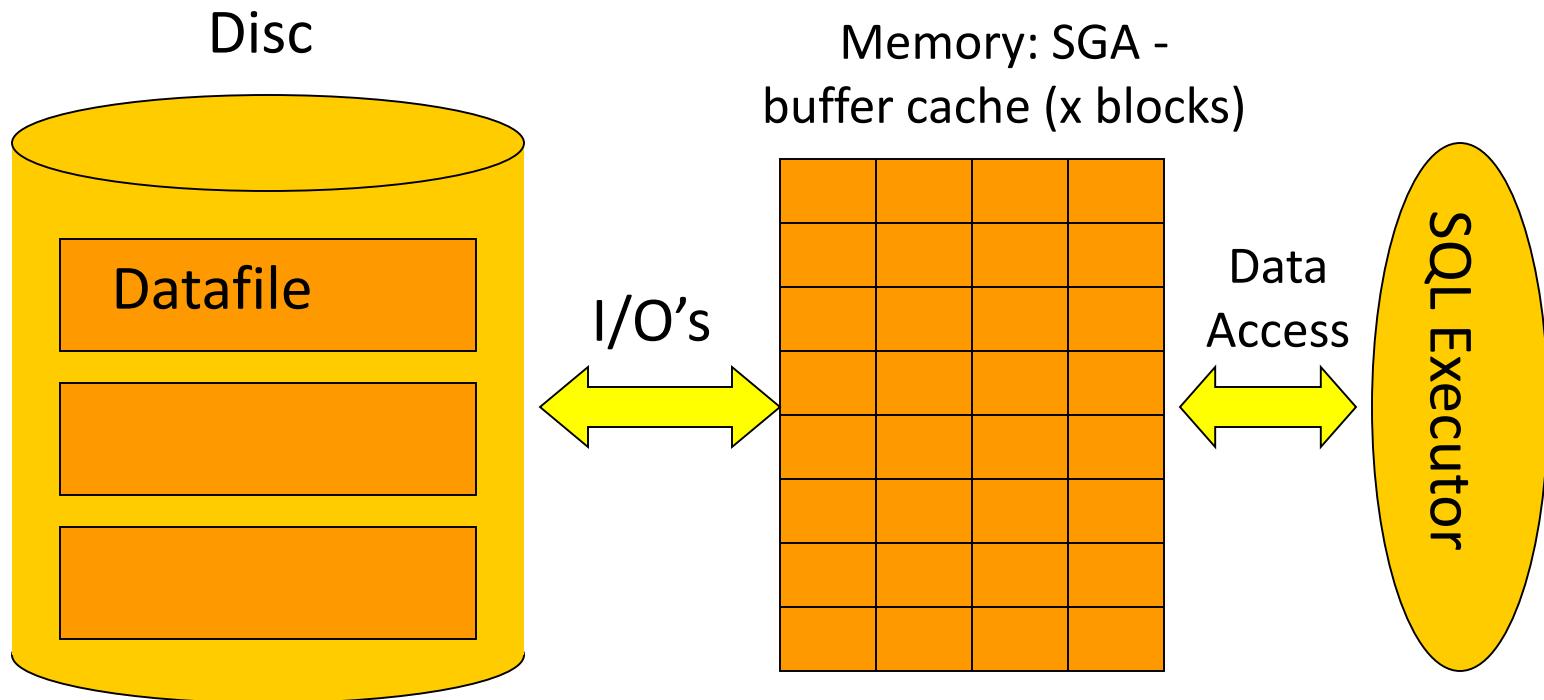
Datafile

Block 1	Block 2	Block 3	Block 4
Block 5	Block ...	Index Node Block	Index Leaf Block
Index Leaf Block			

No particular order of node and leaf blocks

Table & Index I/O

- I/O's are done at 'block level'
 - LRU list controls who 'makes place' in the cache

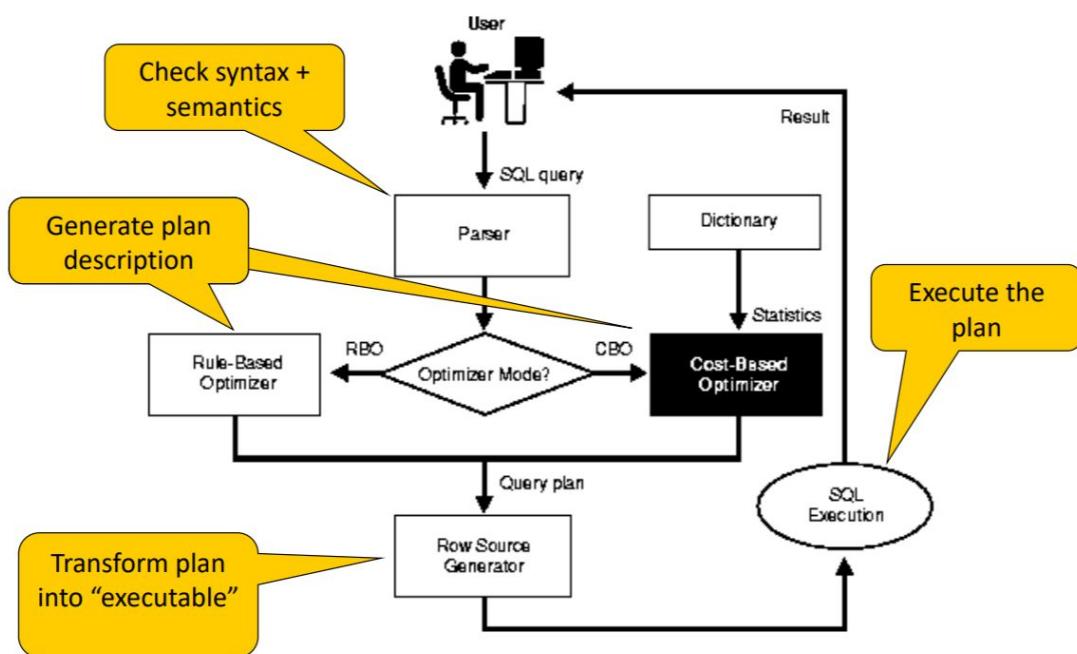


WEEK 5 - 9ituning - Original Page 2

Optimizer Overview

User -> SQL query in Parser (Check syntax semantics) ->
Rule-Based Optimizer RBO (Generate plan description)<-
Optimizer Mode -> CBO Cost-Based Optimizer (Generate
plan description) <- Dictionary (Statistics)
-> Row Source Generator (Query Plan) (Transform plan
into “executable”) -> SQL Execution (Execute the plan) ->
User (Result)

Optimizer Overview



On-Line Analytic Processing

Warehousing

Data Cubes

Data Mining

Based on Molnár Bálint's
lecture

Overview

- Traditional database systems are tuned to many, small, simple queries.
- Some new applications use fewer, more time-consuming, complex queries.
- New architectures have been developed to handle complex “analytic” queries efficiently.

The Data Warehouse

- The most common form of data integration.
 - Copy sources into a single DB (*warehouse*) and try to keep it up-to-date.
 - Usual method: periodic reconstruction of the warehouse, perhaps overnight.
 - Frequently essential for analytic queries.

OLTP

- Most database operations involve *On-Line Transaction Processing* (OLTP).
- Short, simple, frequent queries and/or modifications, each involving a small number of tuples.
- Examples: Answering queries from a Web interface, sales at cash registers, selling airline tickets.

OLAP

- Of increasing importance are *On-Line Analytic Processing* (OLAP) queries.
 - Few, but complex queries --- may run for hours.
 - Queries do not depend on having an absolutely up-to-date database.
- Sometimes called *Data Mining*.

OLAP Examples

1. Amazon analyzes purchases by its customers to come up with an individual screen with products of likely interest to the customer.
2. Analysts at Wal-Mart look for items with increasing sales in some region.

Common Architecture

- Databases at store branches handle OLTP.
- Local store databases copied to a central warehouse overnight.
- Analysts use the warehouse for OLAP.

Star Schemas

- A *star schema* is a common organization for data at a warehouse. It consists of:
 1. *Fact table* : a very large accumulation of facts such as sales.
 - Often “insert-only.”
 2. *Dimension tables* : smaller, generally static information about the entities involved in the facts.

Example: Star Schema

- Suppose we want to record in a warehouse information about every beer sale: the bar, the brand of beer, the drinker who bought the beer, the day, the time, and the price charged.
- The fact table is a relation:
 $\text{Sales}(\text{bar}, \text{beer}, \text{drinker}, \text{day}, \text{time}, \text{price})$

Example, Continued

- The dimension tables include information about the bar, beer, and drinker “dimensions”:
- Bars(bar, addr, license)
- Beers(beer, manf)
- Drinkers(drinker,addr, phone)

Dimensions and Dependent Attributes

- Two classes of fact-table attributes:
 1. *Dimension attributes* : the key of a dimension table.
 2. *Dependent attributes* : a value determined by the dimension attributes of the tuple.

Example: Dependent Attribute

- *price* is the dependent attribute of our example Sales relation.
- It is determined by the combination of dimension attributes: *bar*, *beer*, *drinker*, and the time (combination of *day* and *time* attributes).

Approaches to Building Warehouses

1. *ROLAP* = “relational OLAP”: Tune a relational DBMS to support star schemas.
2. *MOLAP* = “multidimensional OLAP”: Use a specialized DBMS with a model such as the “data cube.”

ROLAP Techniques

1. *Bitmap indexes* : For each key value of a dimension table (e.g., each beer for relation Beers) create a bit-vector telling which tuples of the fact table have that value.
2. *Materialized views* : Store the answers to several useful queries (views) in the warehouse itself.

Typical OLAP Queries

- Often, OLAP queries begin with a “star join”: the natural join of the fact table with all or most of the dimension tables.

```
SELECT *
FROM Sales, Bars, Beers, Drinkers
WHERE Sales.bar = Bars.bar AND
      Sales.beer = Beers.beer AND
      Sales.drinker = Drinkers.drinker;
```

Typical OLAP Queries --- 2

- The typical OLAP query will:
 1. Start with a star join.
 2. Select for interesting tuples, based on dimension data.
 3. Group by one or more dimensions.
 4. Aggregate certain attributes of the result.

Example: OLAP Query

- For each bar in Palo Alto, find the total sale of each beer manufactured by Anheuser-Busch.
1. Filter: $addr = \text{"Palo Alto"}$ and $manf = \text{"Anheuser-Busch"}$.
 2. Grouping: by bar and beer.
 3. Aggregation: Sum of $price$.

Example: In SQL

```
SELECT bar, beer, SUM(price)
FROM Sales NATURAL JOIN Bars
      NATURAL JOIN Beers
WHERE addr = 'Palo Alto' AND
      manf = 'Anheuser-Busch'
GROUP BY bar, beer;
```

Using Materialized Views

- A direct execution of this query from *Sales* and the dimension tables could take too long.
- If we create a materialized view that contains enough information, we may be able to answer our query much faster.

Example: Materialized View

- Which views could help with our query?
- Key issues:
 1. It must join Sales, Bars, and Beers, at least.
 2. It must group by at least bar and beer.
 3. It must not select out Palo-Alto bars or Anheuser-Busch beers.
 4. It must not project out *addr* or *manf*.

Example --- Continued

Here is a materialized view
that could help:

```
CREATE VIEW BABMS(bar, addr,  
                  beer, manf, sales) AS  
  
SELECT bar, addr, beer, manf,  
      SUM(price) sales  
  
FROM Sales NATURAL JOIN Bars  
      NATURAL JOIN Beers  
      GROUP BY bar, addr, beer, manf;
```

Since $\text{bar} \rightarrow \text{addr}$ and $\text{beer} \rightarrow \text{manf}$, there is no real
grouping. We need addr and manf in the SELECT.

Example --- Concluded

Here's our query using the materialized view BABMS:

```
SELECT bar, beer, sales  
FROM BABMS  
WHERE addr = 'Palo Alto' AND  
      manf = 'Anheuser-Busch';
```

MOLAP and Data Cubes

- Keys of dimension tables are the dimensions of a hypercube.
 - Example: for the Sales data, the four dimensions are bars, beers, drinkers, and time.
- Dependent attributes (e.g., price) appear at the points of the cube.

Marginals

- The data cube also includes aggregation (typically SUM) along the margins of the cube.
- The marginals include aggregations over one dimension, two dimensions,...

Example: Marginals

- Our 4-dimensional Sales cube includes the sum of *price* over each bar, each beer, each drinker, and each time unit (perhaps days).
- It would also have the sum of *price* over all bar-beer pairs, all bar-drinker-day triples,...

Structure of the Cube

- Think of each dimension as having an additional value *.
- A point with one or more *'s in its coordinates aggregates over the dimensions with the *'s.
- Example: Sales("Joe's Bar", *, "Bud", *, *sales*) holds the sum over all drinkers and all time of the Bud consumed at Joe's.

Sale(bar, addr, beer, manf, sales)

- Pl. Sales("Joe's Bar", *,
"Bud", *, *sales*)

Drill-Down

- *Drill-down* = “de-aggregate” = break an aggregate into its constituents.
- Example: having determined that Joe’s Bar sells very few Anheuser-Busch beers
- Break down his sales by particular A.-B. beer.

Roll-Up

- *Roll-up* = aggregate along one or more dimensions.
- Example: given a table of how much Bud each drinker consumes at each bar
- Roll it up into a table giving total amount of Bud consumed for each drinker.

Materialized Data-Cube Views

- Data cubes invite materialized views that are aggregations in one or more dimensions.
- Dimensions may not be completely aggregated --- an option is to group by an attribute of the dimension table.

Example

- A materialized view for our Sales data cube might:
 1. Aggregate by drinker completely.
 2. Not aggregate at all by beer.
 3. Aggregate by time according to the week.
 4. Aggregate according to the city of the bar.

Data Mining

- *Data mining* is a popular term for queries that summarize big data sets in useful ways.
- Examples:
 1. Clustering all Web pages by topic.
 2. Finding characteristics of fraudulent credit-card use.

Market-Basket Data

- An important form of mining from relational data involves *market baskets* = sets of “items” that are purchased together as a customer leaves a store.
- Summary of basket data is *frequent itemsets* = sets of items that often appear together in baskets.

Example: Market Baskets

- If people often buy hamburger and ketchup together, the store can:
 1. Put hamburger and ketchup near each other and put potato chips between.
 2. Run a sale on hamburger and raise the price of ketchup.

Finding Frequent Pairs

- The simplest case is when we only want to find “frequent pairs” of items.
- Assume data is in a relation *Baskets(basket, item)*.
- The *support threshold s* is the minimum number of baskets in which a pair appears before we are interested.

Frequent Pairs in SQL

Search for those basket pairs, where the baskets are the same, but the items are different.

```
SELECT b1.item, b2.item
```

```
FROM Baskets b1, Baskets b2
```

```
WHERE b1.basket = b2.basket
```

```
AND b1.item < b2.item
```

The first has to precede the other, So that we do not count the same twice

```
GROUP BY b1.item, b2.item
```

```
HAVING COUNT(*) >= s;
```

Throw away pairs of items that do not appear at least s times.

Create a group for each pair of items that appears in at least one basket.

A-Priori Trick --- 1

- Straightforward implementation involves a join of a huge *Baskets* relation with itself.
- The *a-priori algorithm* speeds the query by recognizing that a pair of items $\{i,j\}$ cannot have support s unless both $\{i\}$ and $\{j\}$ do.

A-Priori Trick --- 2

- Use a materialized view to hold only information about frequent items.

```
INSERT INTO Baskets1(basket, item)
SELECT * FROM Baskets
WHERE item IN (
    SELECT ITEM FROM Baskets
    GROUP BY item
    HAVING COUNT(*) >= s
);
```

Items that appear in at least s baskets.

A-Priori Algorithm

A-Priori Algoritmus

1. Materialize the view *Baskets1*.
2. Run the obvious query, but on *Baskets1* instead of *Baskets*.
 - *Baskets1* is cheap, since it doesn't involve a join.
 - *Baskets1* *probably* has many fewer tuples than *Baskets*.
 - Running time shrinks with the *square* of the number of tuples involved in the join.

NoSql, Document Store, MongoDB

Lehotay-Kéry Péter
based on lecture of
Gombos Gergő

Overview

- MongoDB
 - Concept
 - JSON, BSON
 - Data storage
 - Replica, sharding
 - Mongo CRUD
 - Aggregation framework

MongoDB=„document oriented”

- Storing documents independently from language in BSON format.

```
{
```

```
  x : 3,
```

```
  y : “abc”,
```

```
  z : [1,2],
```

```
  d : { }
```

```
}
```

JSON

- Serves to store structured documents.
- Similar document format: eg XML
- Data types:
 - string
 - number
 - boolean
 - NULL
 - array
 - object/document
- Associative array: only string can be the key

```
{  
  "name": "jill",  
  "age": 2,  
  "voted": true,  
  "school": null,  
  "likes": ["tennis", "math"],  
  "addr": {  
    "city": "New York",  
    "addr": "3rd Street"  
  }  
}
```

XML vs JSON

```
<phones>
    <phone type =“home”>123</phone>
    <phone type =“mobile”>456</phone>
</phones>
```

```
{
    “phones”: [
        {"phone": 123, "type": "home"},
        {"phone": 456, "type": "mobil"}
    ]
}
```

BSON

- Binary representation of JSON
- Easily scalable
- Mongoimport for BSON operations
- JSON:

```
{  
  „a”: 3,  
  „b”:”xyz”  
}
```

- BSON:

23	\x10	a \0	3	\x02	b \0	x y z \0	\0
----	------	------	---	------	------	----------	----

Doc length	key type(int)	value	type(string)	key	value	End of doc
23	\x10	a \0	3	\x02	b \0	x y z \0

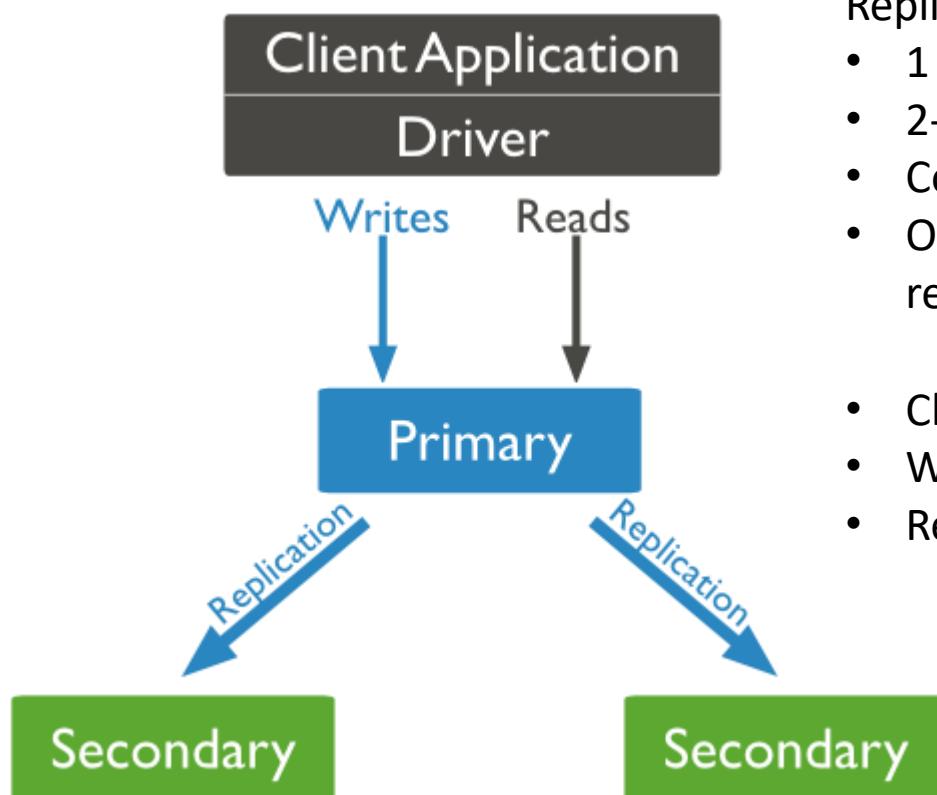
Schemaless

- Example collection:
 - {shape: “rectangle”, x: 3, y:4, “area”: 12}
 - {shape: “circle”, r: 1, “area”: 3.14}
 - {q: 456}
- In practice, structure is consistent!
 - { name : “Joe”, age : 30, interests : ‘football’ }
 - { name : “Kate”, age : 25 }

MongoDB elements

RDBMS	MongoDB
Database	Database
Table	Collection
Row	Document (JSON,BSON)
Column	Field
Index	Index
Join	Embedded Document
Foreign Key	Reference
Queries return records	Queries return a cursor

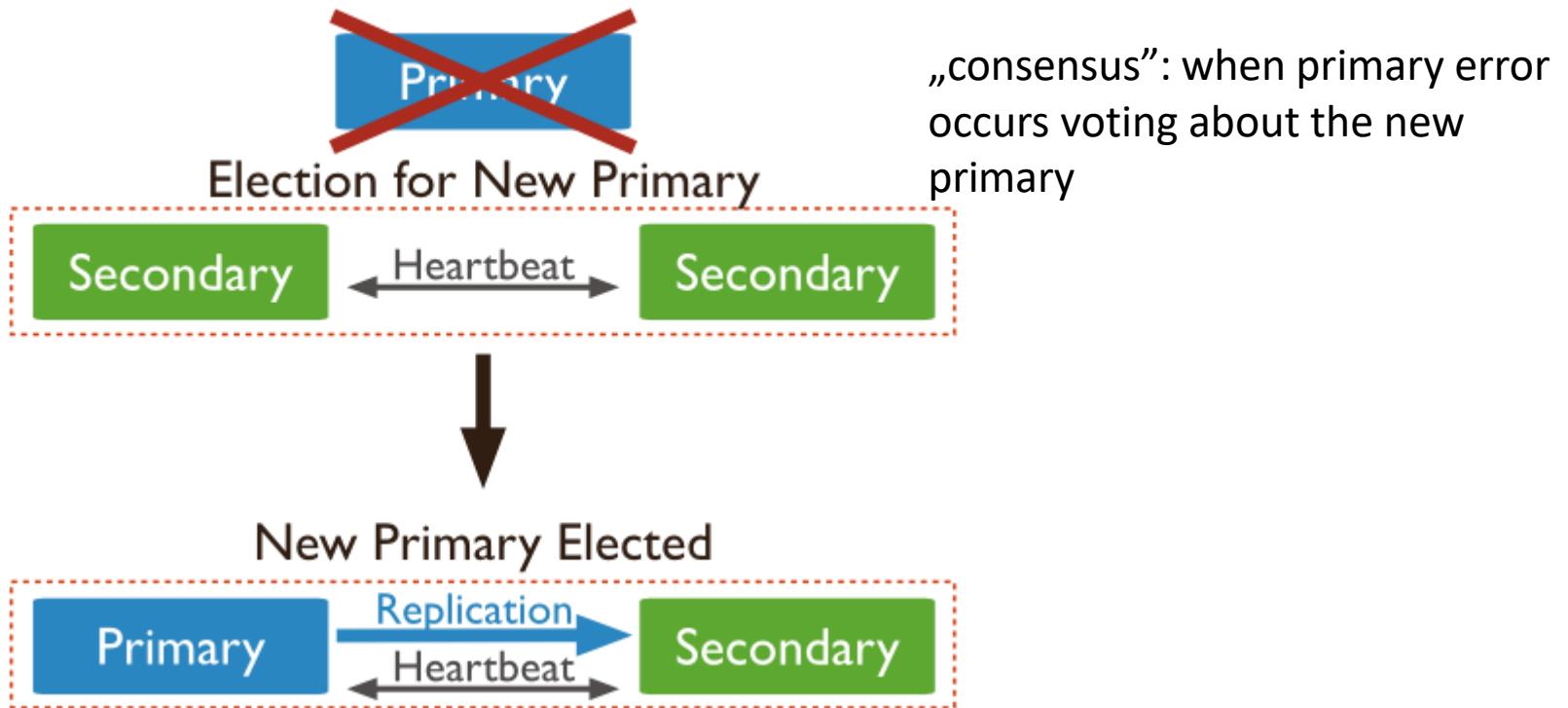
MongoDB high availability



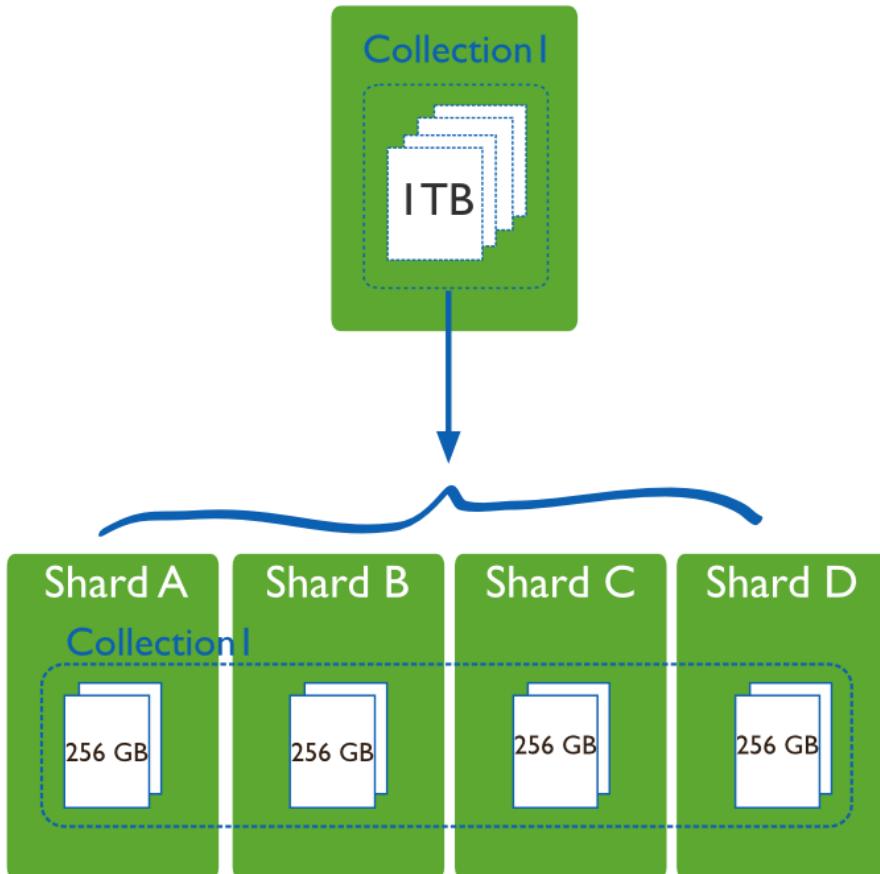
Replica Set

- 1 Primary
- 2-48 Secondaries
- Copies from original data
- Operations affecting multiple rows replicate row by row
- Client always connects on primary
- Writing always goes on primary
- Reading can go on secondary

MongoDB high availability



MongoDB high availability

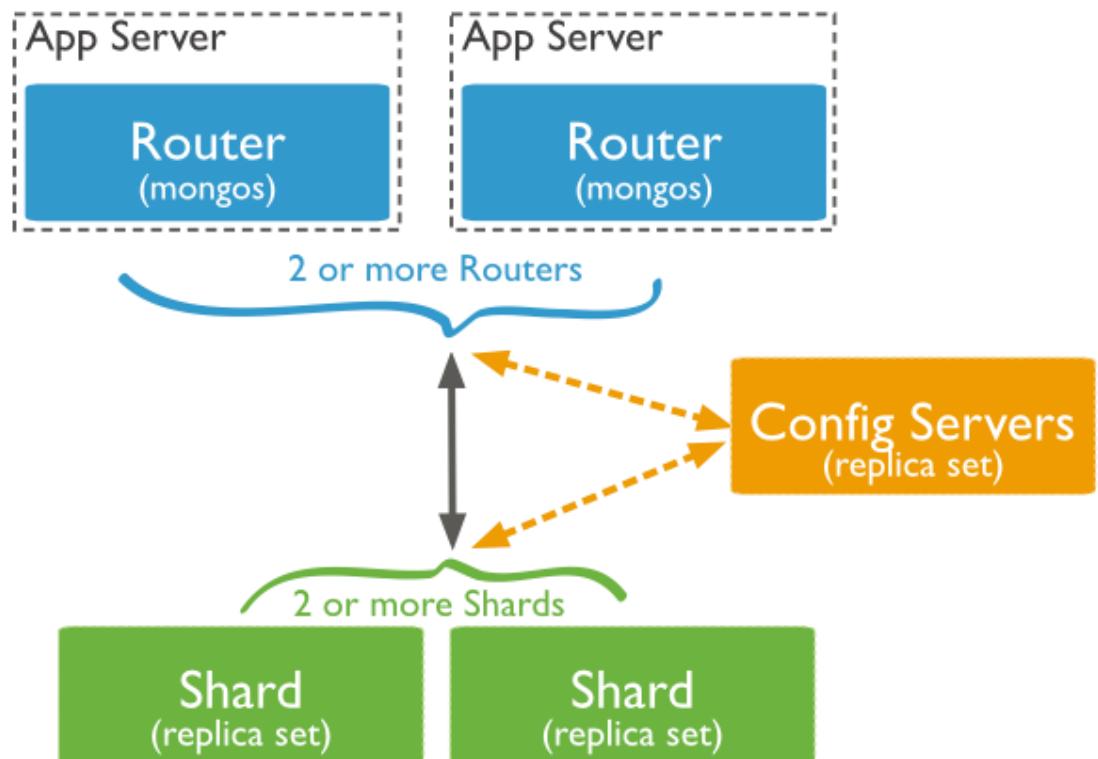


Sharding

- Collections distributed in cluster based on shard key
- shard key = indexed data
- Inside shard can have replicates
- Lowering the load on servers
 - Querying on multiple machines
 - Fewer data on a single machine

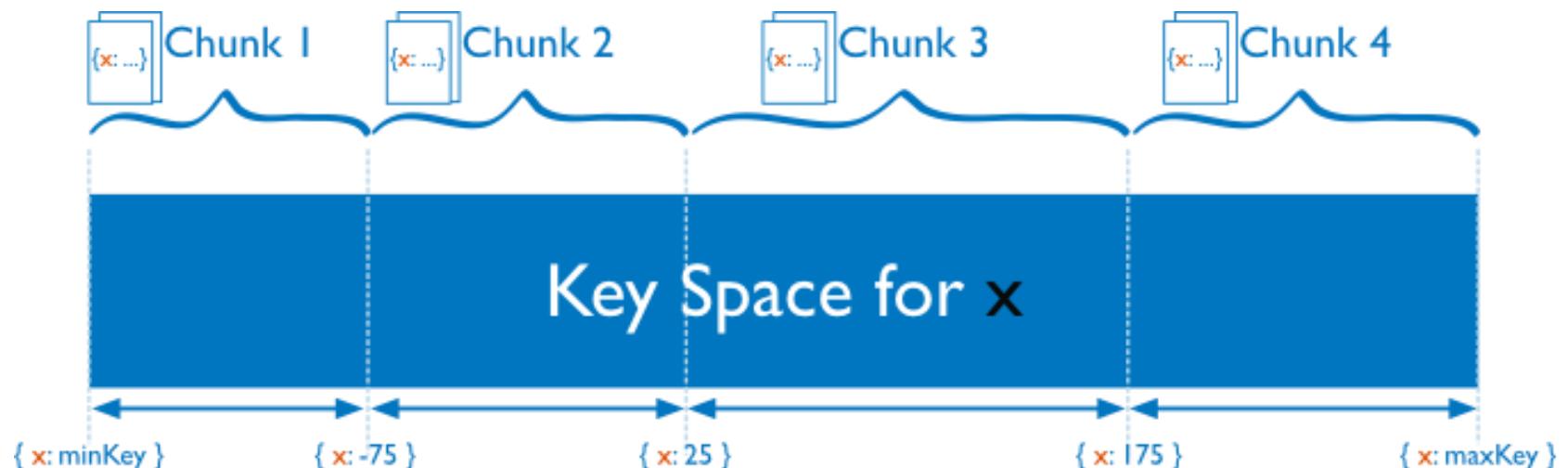
MongoDB sharding components

- shards
 - Storing the data
- routers
 - interface between client and shard
- config server
 - metadata about the data



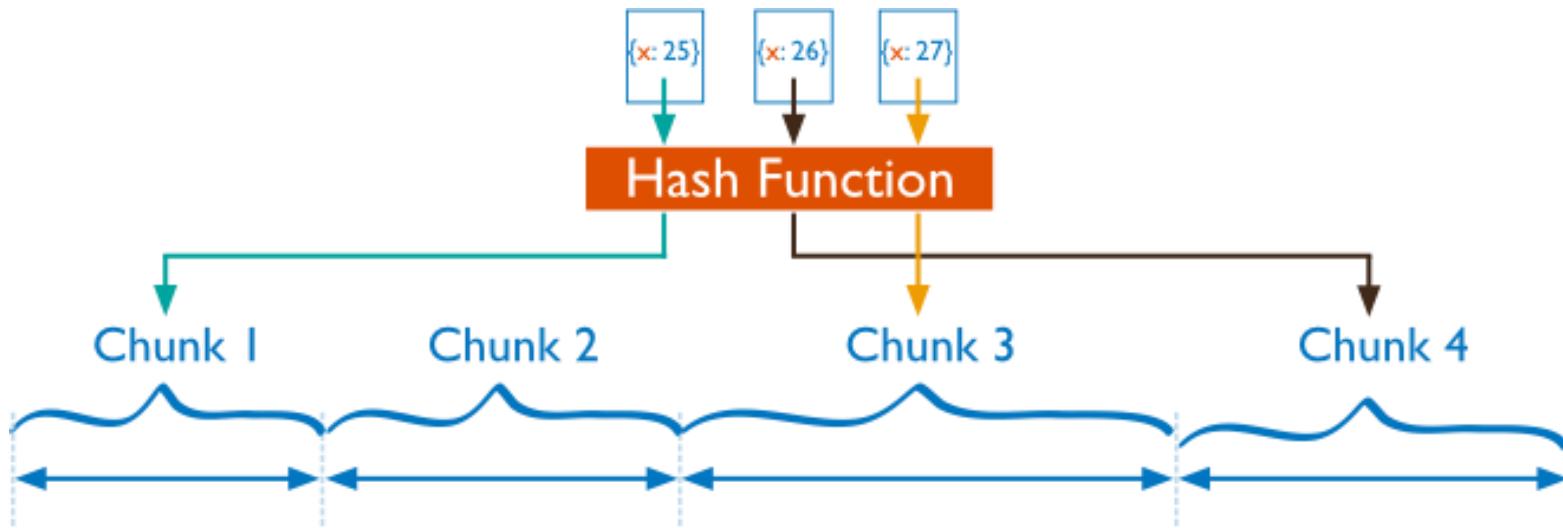
Sharding solutions

- Range based sharding
 - shard key storing based on domains
 - Efficient domain queries



Sharding solutions

- Hash based sharding
 - distributing data based on given hash function



Data insertion

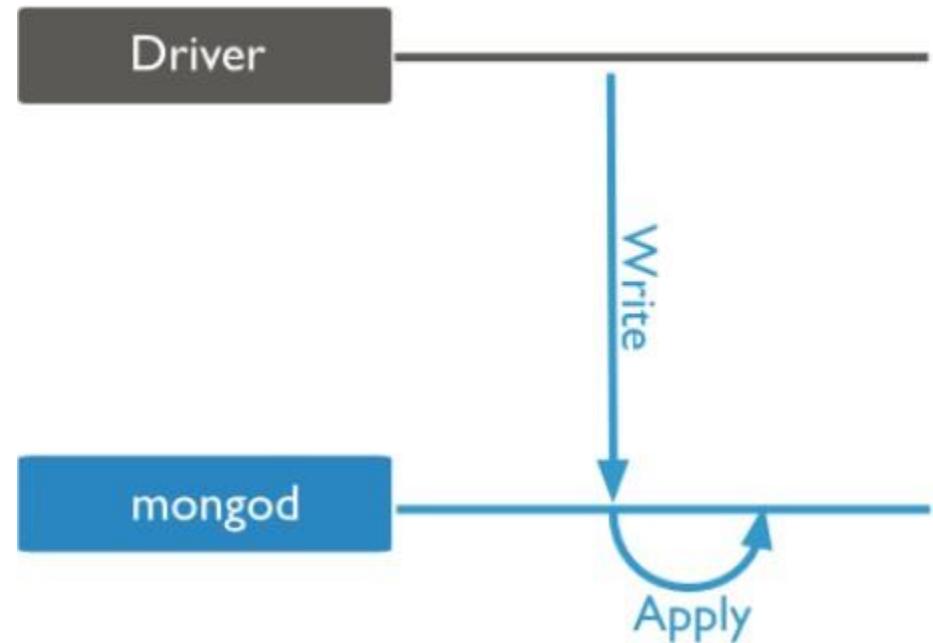
- „write concern”
- At insert, update, delete statement:
 - Strong supervision = slow answer
 - Weak supervision = fast answer
- MongoDB can be parameterised

Levels of „write concern”

- w and j parameters can be set!
- Unacknowledged
- Acknowledged (w:1)
- Journalized (j:true)
- Replica Set Acknowledged (w:2)

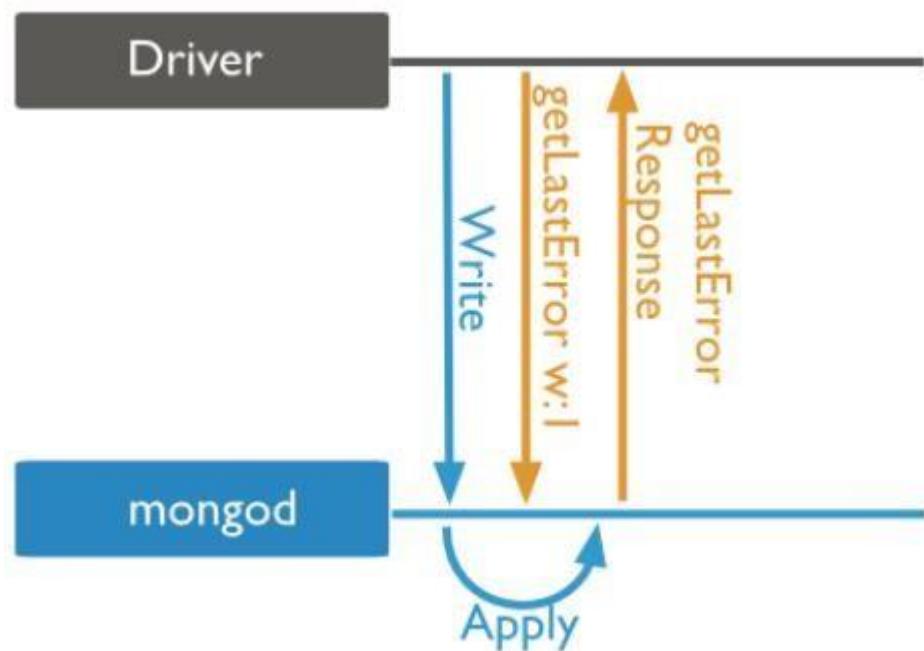
Unacknowledged

- Client does not get feedback of writing
- Very fast writing
- Possible data loss!
 - Networking error
 - Key collision



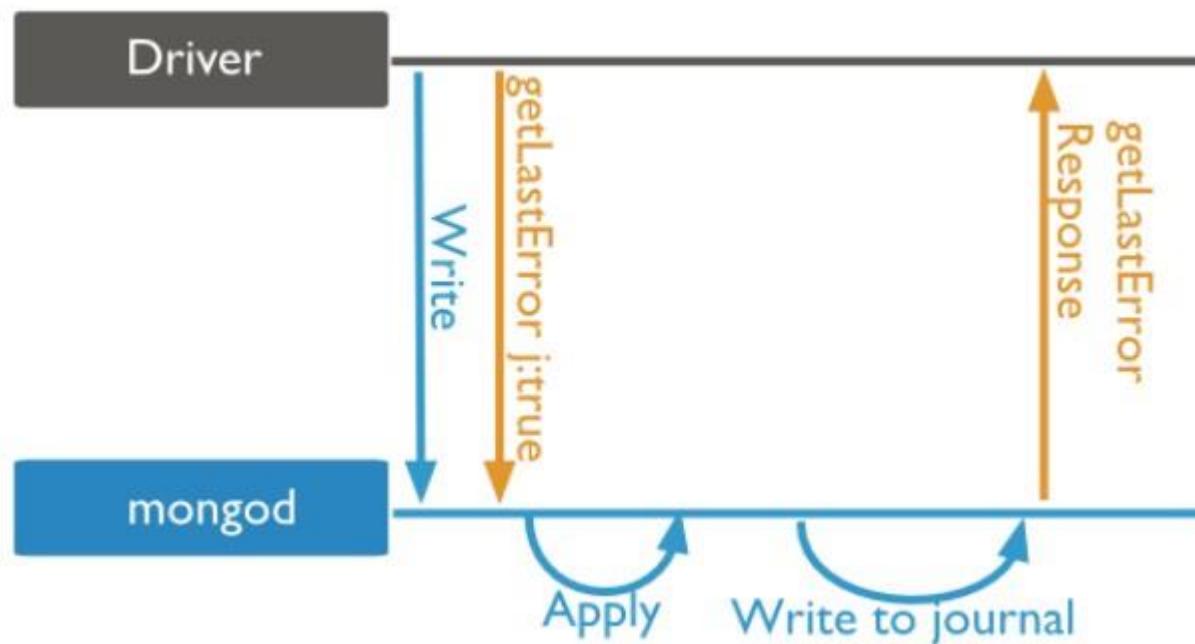
Acknowledged

- Mongod sends receipt
- Client detects errors (networking, key collusion)
- This is the default value
- Writing is not guaranteed!



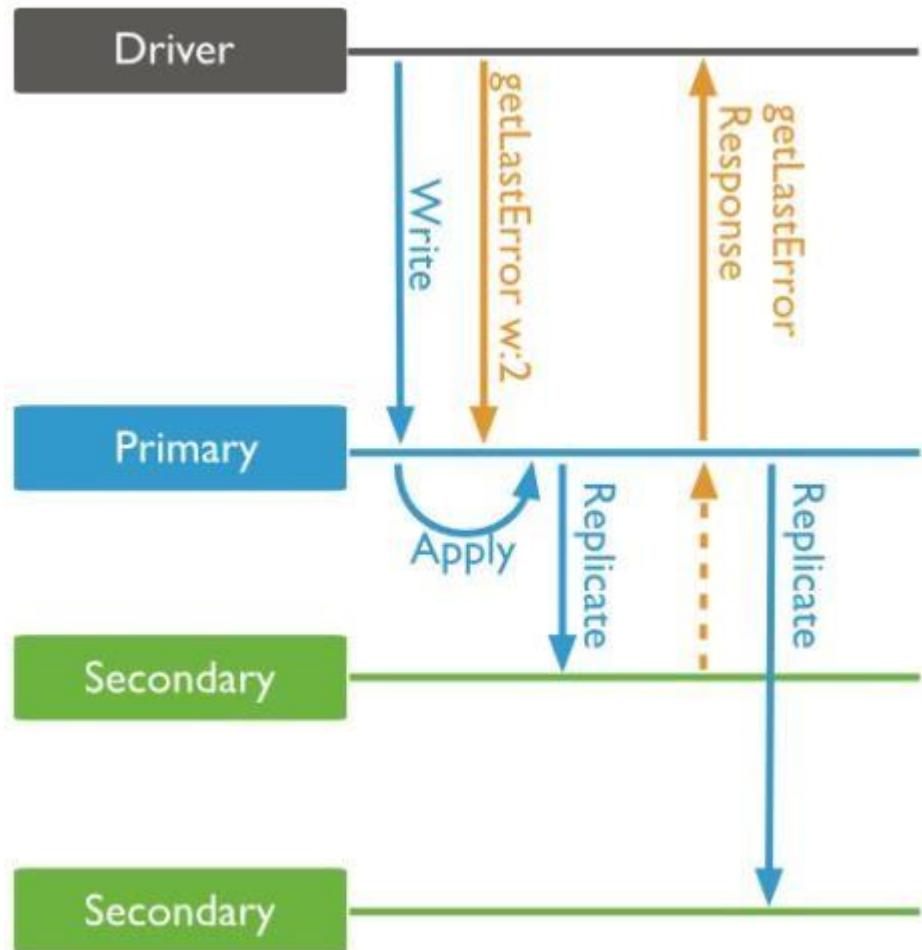
Jounaled

- Mongod does not acknowledges, until it does not write into the journal. (logging)
- Journal have to be enabled



Replica Set Acknowledged

- Can be set, how many replicas have to confirm the writing before it is noted to the client



MongoDB data storage

- Every MongoDB instance includes:
 - Namespace file
 - Journal file
 - Data file
- Data file stores in extents:
 - BSON documents
 - Indices
 - MongoDB metadata data
- Extent: logical container

Extents

my-db.1

my-db.2



- Data and index separate extent
- An extent corresponds to a collection
- A collection can be in multiple extent



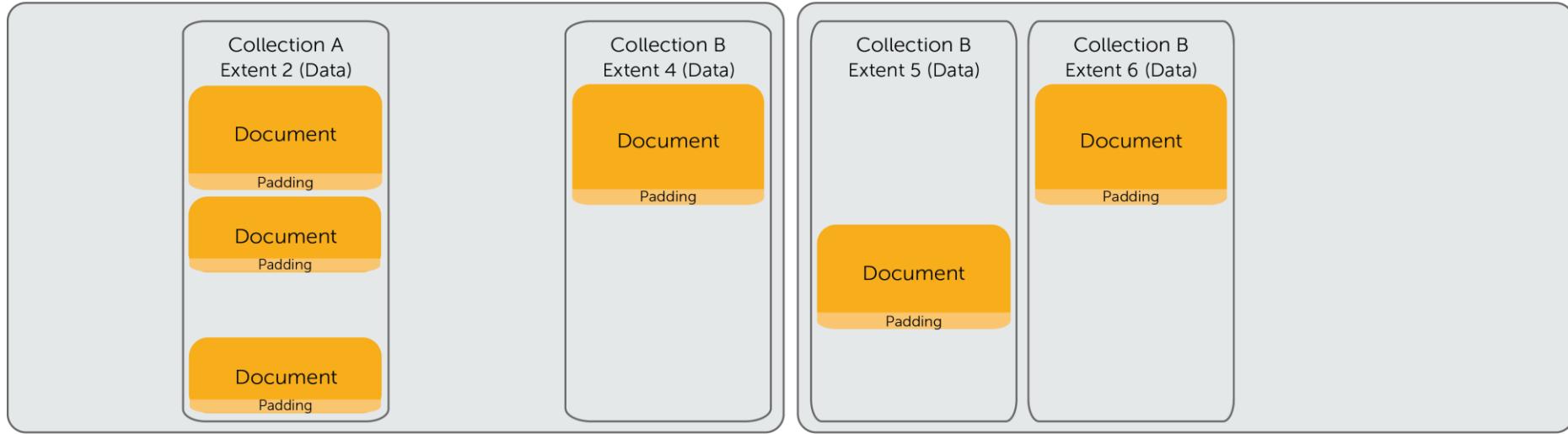
Metrics

- db.stats()
 - Statistics about our database
 - dataSize
 - storageSize
 - fileSize

dataSize

my-db.1

my-db.2



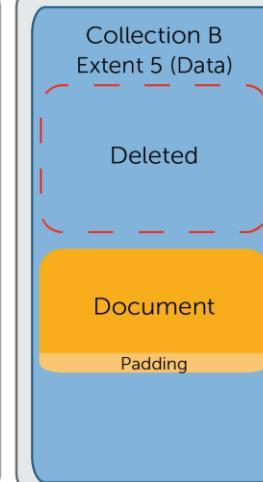
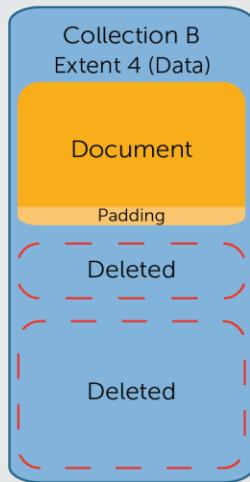
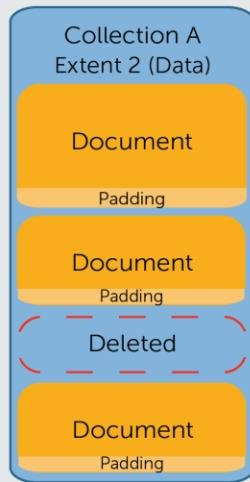
- **dataSize** is the sum of documents and reserved fields in byte
- If the data changes but fits in the reserved area then the **dataSize** is not changing

dataSize

my-db.1

storageSize

my-db.2



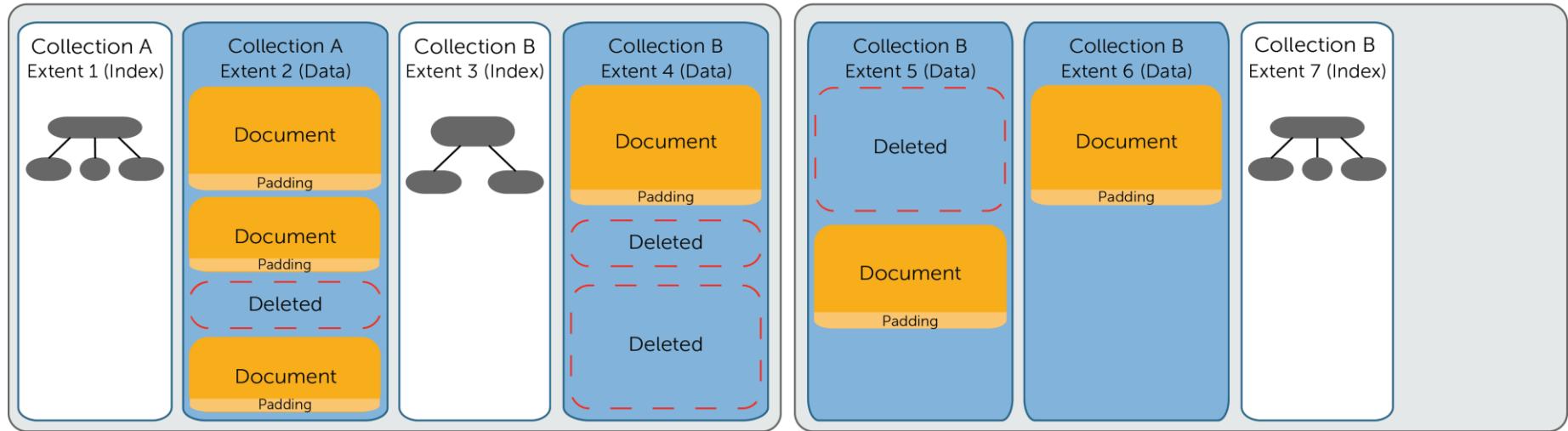
■ dataSize
■ storageSize

- Size of data extents
- Including the unused areas

fileSize

my-db.1

my-db.2



- Data extents, index extents and the unused areas in the file
- The full reserved area on the drive

Mongo shell

- Available: Windows, Linux, MacOS
- importing data
 - mongoimport.exe --db test --collection restaurants --drop --file mongo_restaurant_dataset.json
- Starting mongo shell
 - mongo.exe

Mongo shell

- Listing databases
 - show dbs
- Listing collections
 - show collections
- Selecting database
 - use test

Mongo CRUD

- Create
 - db.collection.insert(<document>)
 - db.collection.update(<query>, <update>, { upsert: true })
- Read
 - db.collection.find(<query>, <projection>)
 - db.collection.findOne(<query>, <projection>)
- Update
 - db.collection.update(<query>, <update>, <options>)
- Delete
 - db.collection.remove(<query>, <justOne>)

Mongo CRUD

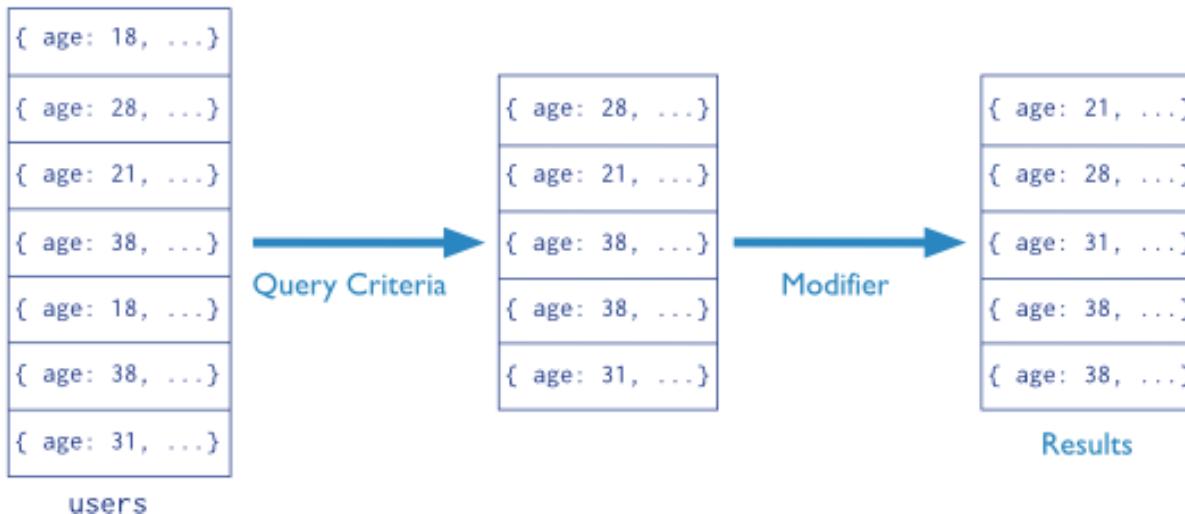
- Listing data
 - db.restaurants.find()
- How many elements?
 - db.restaurants.count()
- Which restaurants are in Manhattan?
 - db.restaurants.find({borough:"Manhattan"})

Mongo CRUD

```
db.users.find(  
  { age: { $gt: 18 } },  
  { name: 1, address: 1 }  
).limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

Collection	Query Criteria	Modifier
db.users	.find({ age: { \$gt: 18 } })	.sort({age: 1})



restaurant example

```
{  
    "_id" : ObjectId("56ed68032ea3567be5c25d32"),  
    "address" : {  
        "building" : "284",  
        "coord" : [ -73.9829239, 40.6580753 ],  
        "street" : "Prospect Park West",  
        "zipcode" : "11215"  
    },  
    "borough" : "Brooklyn",  
    "cuisine" : "American ",  
    "grades" : [  
        { "date" : ISODate("2014-11-19T00:00:00Z"), "grade" : "A", "score" : 11 },  
        { "date" : ISODate("2013-11-14T00:00:00Z"), "grade" : "A", "score" : 2 },  
        { "date" : ISODate("2012-12-05T00:00:00Z"), "grade" : "A", "score" : 13 },  
        { "date" : ISODate("2012-05-17T00:00:00Z"), "grade" : "A", "score" : 11 }  
    ],  
    "name" : "The Movable Feast",  
    "restaurant_id" : "40361606"  
}
```

Mongo CRUD

- Filtering for embedded object
 - db.restaurants.find({ "address.zipcode": "10075" })
- Searching in array
 - db.restaurants.find({ "grades.grade": "B" })
- Using operators
 - db.restaurants.find({ "grades.score": { \$gt: 30 } })
 - db.restaurants.find({ "grades.score": { \$lt: 10 } })

Mongo CRUD

- Connecting conditions
- logical and
 - db.restaurants.find({ "cuisine": "Italian", "address.zipcode": "10075" })
- logical or
 - db.restaurants.find({ \$or: [{ "cuisine": "Italian" }, { "address.zipcode": "10075" }] })
- sorting
 - db.restaurants.find().sort({ "borough": 1, "address.zipcode": 1 })

Mongo CRUD

- update - Updates first object named „Juni”
 - db.restaurants.update(

```
{ "name" : "Juni" },  
{  
  $set: { "cuisine": "American (New)" },  
  $currentDate: { "lastModified": true }  
}  
)
```
- update – inside object
 - db.restaurants.update(

```
{ "restaurant_id" : "41156888" },  
{ $set: { "address.street": "East 31st Street" } }
```

)

Mongo CRUD

- update – refreshing multiple elements
 - db.restaurants.update(

```
{ "address.zipcode": "10016", cuisine: "Other" },  
{  
    $set: { cuisine: "Category To Be Determined" },  
    $currentDate: { "lastModified": true }  
},  
{ multi: true}  
)
```

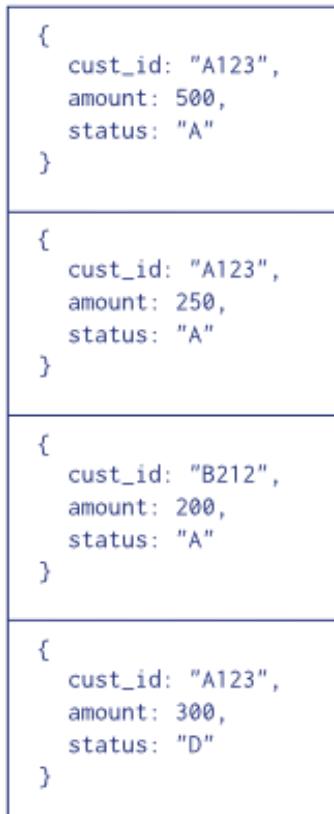
Mongo CRUD

- delete – deletes all corresponding documents
 - db.restaurants.remove({ "borough": "Manhattan" })
- delete – only the first
 - db.restaurants.remove({ "borough": "Queens" },
{ justOne: true })

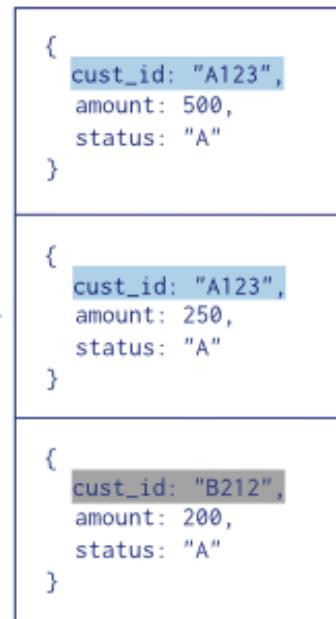
Aggregation framework

Collection
↓

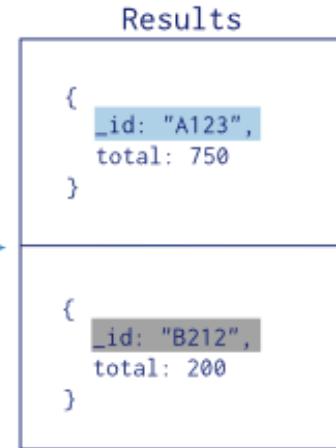
```
db.orders.aggregate( [  
    $match stage → { $match: { status: "A" } },  
    $group stage → { $group: { _id: "$cust_id", total: { $sum: "$amount" } } }  
] )
```



\$match



\$group



- pipeline philosophy, like in unix

orders

Aggregation framework

- operators:
 - \$project - projection
 - \$match - filtering
 - \$limit - first n rows
 - \$skip - Skip n rows
 - \$group - grouping
 - \$sort - ordering
 - \$exists - exists

Aggregation framework

- grouping
 - db.restaurants.aggregate(

```
[  
  { $group: { "_id": "$borough",  
            "count": { $sum: 1 } } }  
]
```

);

Example query

- Hotel database
- Which state is, where the most hotel is, which has no parking and the utilization is more than 50%?

Example query

- Interested in hotels with no parking
- `{$match: {parking: false}},`
- Keeping `_id`, `state` and we introduce with name `pc` a new attribute, which calculated as `free/room` division
- `{$project: {_id:1, state:1, pct: {$divide: ["$free", "$rooms"]}}},`

Example query

- From these, we let go those, where pc is above 0.5
- `{$match: {pct: {$gt: 0.5}}},`
- We group, where the key of grouping is state and the result-records will have another attribute named n, which will be the summing 1 for the records with the given state code, i.e we count these hotels
- `{$group: {_id: "$state", n: {$sum: 1}}},`

Example query

- In descending order by the number of hotels, we are interested in
- `{$sort: {n: -1}},`
- The top 3.
- `{$limit: 3},`

Example query

- db.hotels.aggregate([
 {\$match: {parking: false}},
 {\$project: {_id:1, state:1, pct: {\$divide:
 ["\$free", "\$rooms"]}}},
 {\$match: {pct: {\$gt: 0.5}}},
 {\$group: {_id: "\$state", n: {\$sum: 1}}},
 {\$sort: {n: -1}},
 {\$limit: 3},])

Graph databases

NoSql, neo4j

Lehotay-Kéry Péter
based on lecture
of Gombos Gergő

Overview

- Why do we use graphs?
- What is a graph?
- How do we work with graph databases (neo4j)



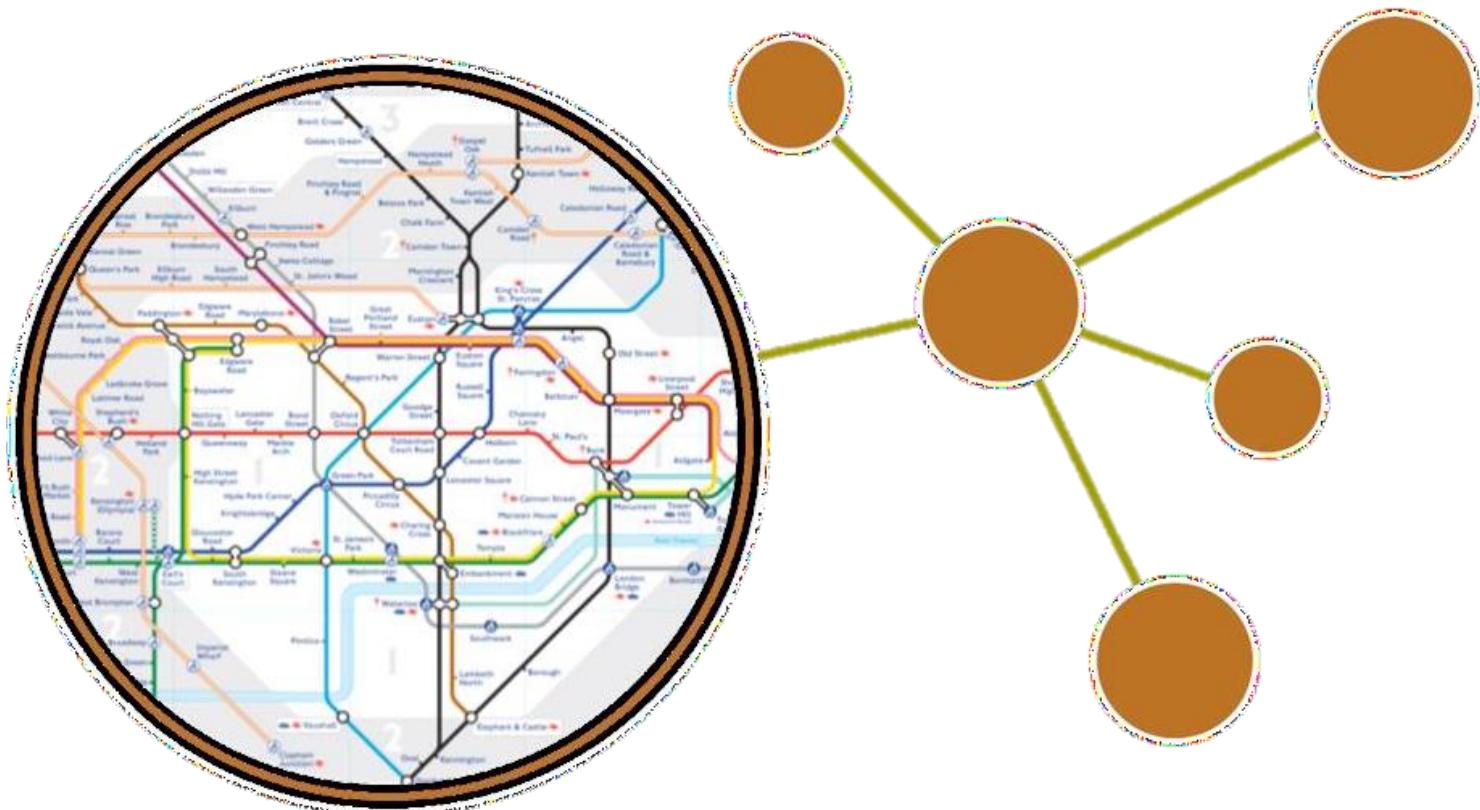
Graph is the whole world

Social networks



Graph is the whole world

- Planning route



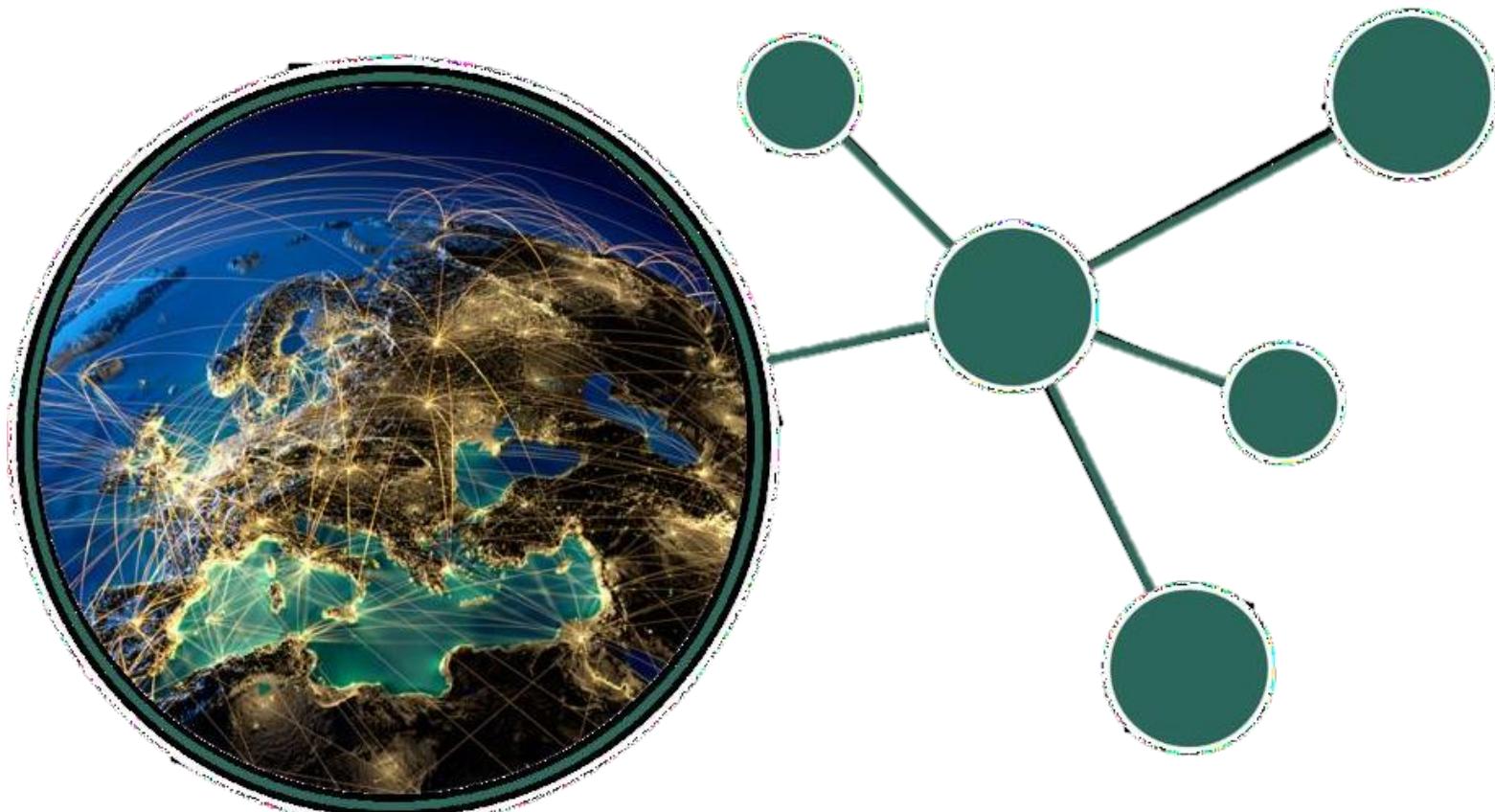
Graph is the whole world

- Recommending systems



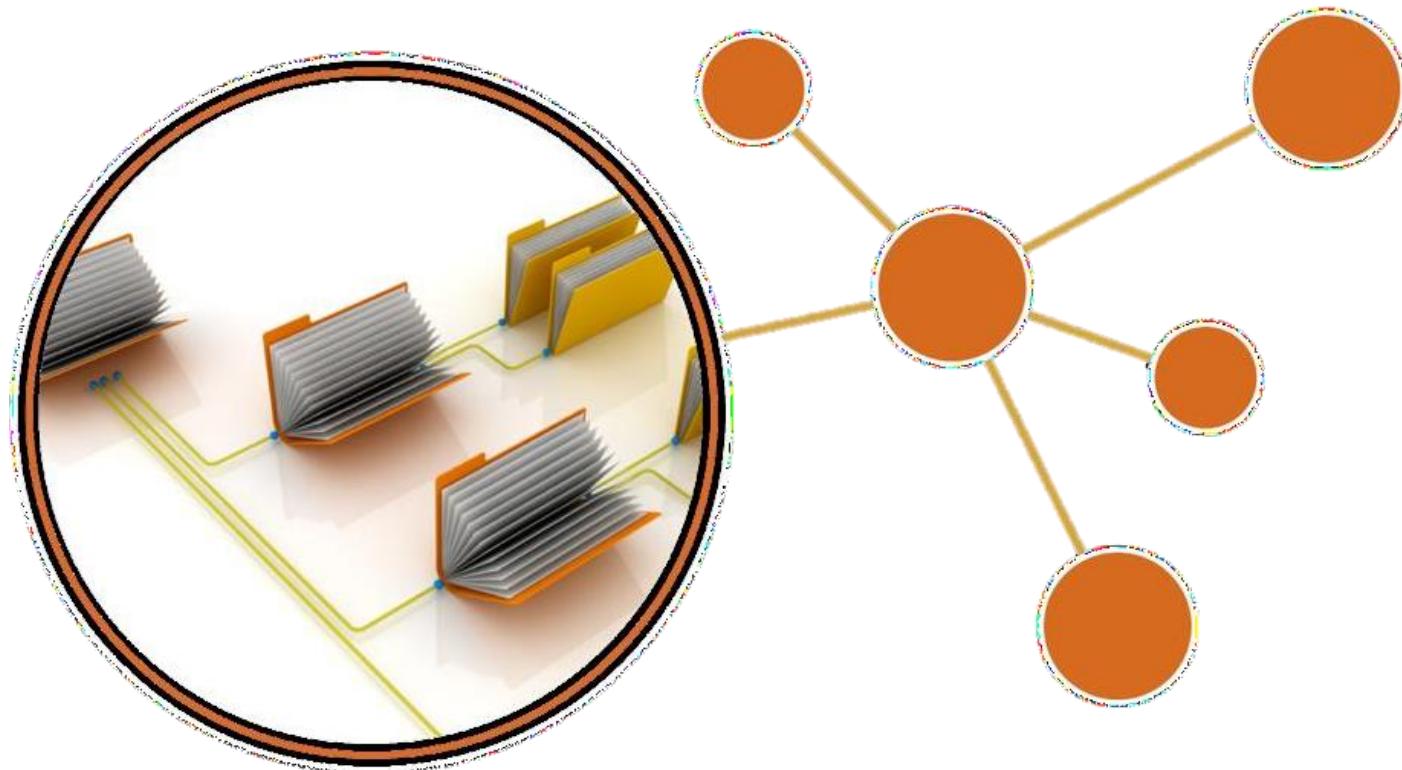
Graph is the whole world

- Logistics

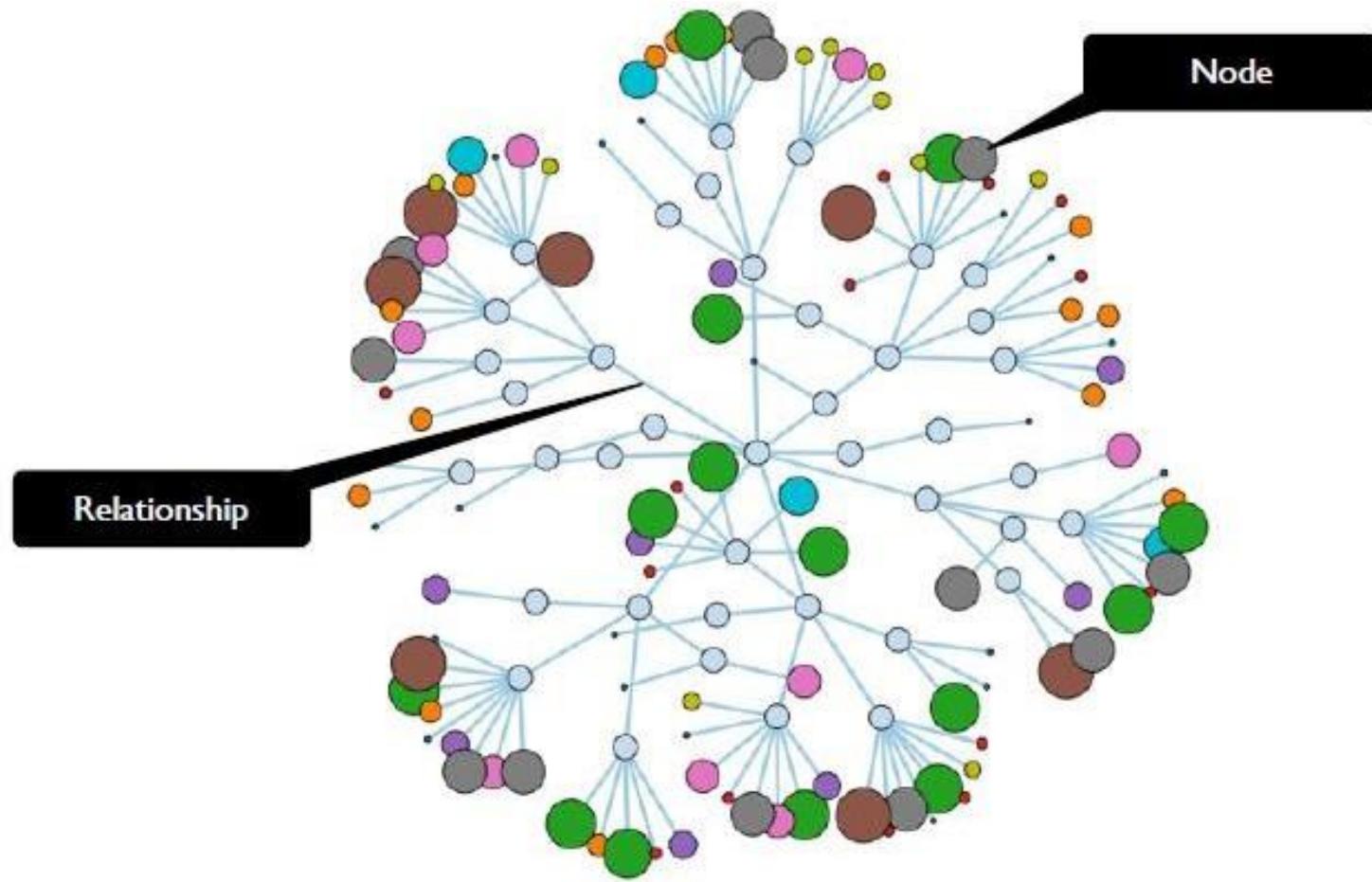


Graph is the whole world

- Permissions



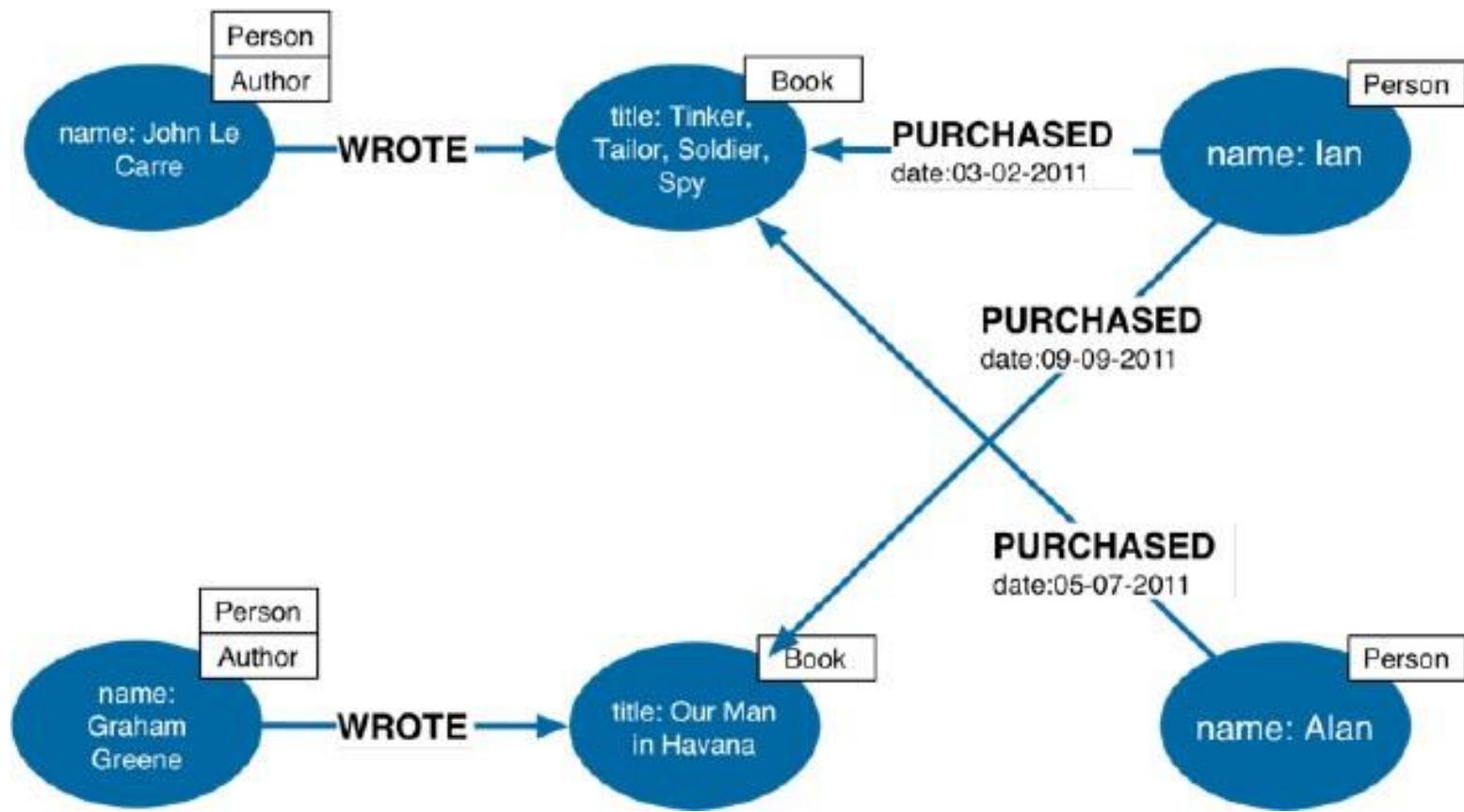
What is the graph?



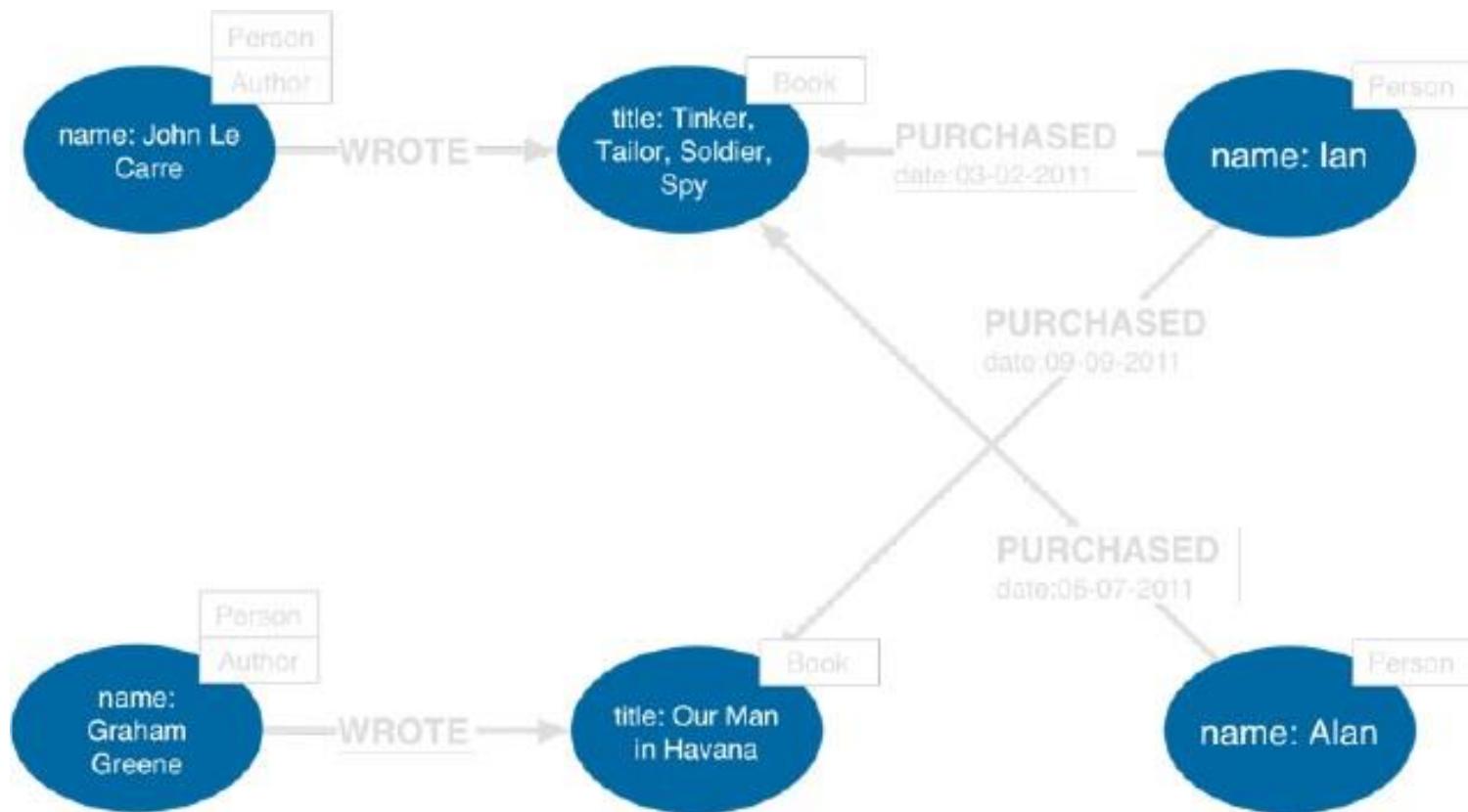
Property graphs

- Building elements:
 - Nodes
 - Entities
 - Edges
 - Relationship between nodes
 - Properties
 - Attributes and metadata
 - Labels
 - grouping

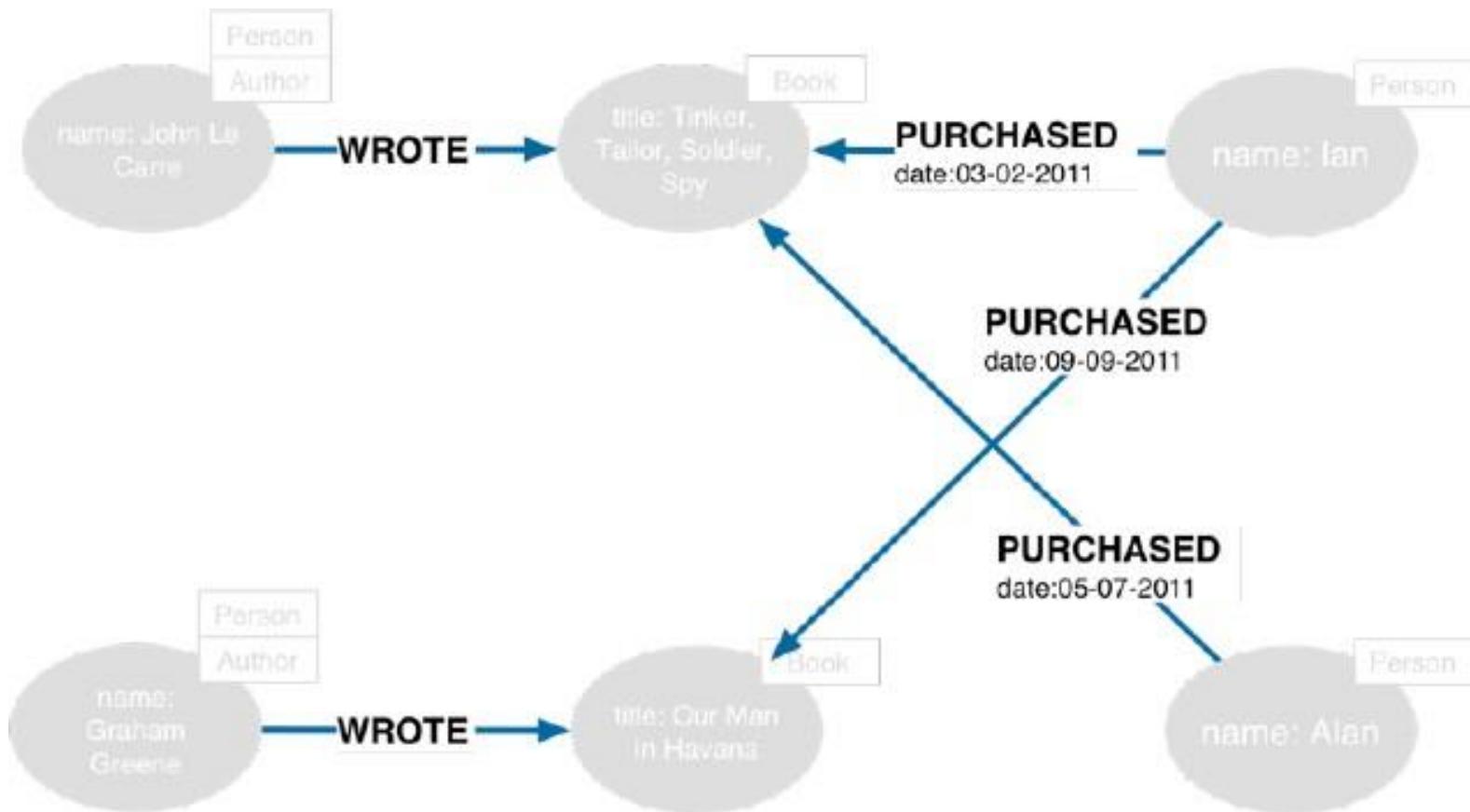
Example property graph



Nodes

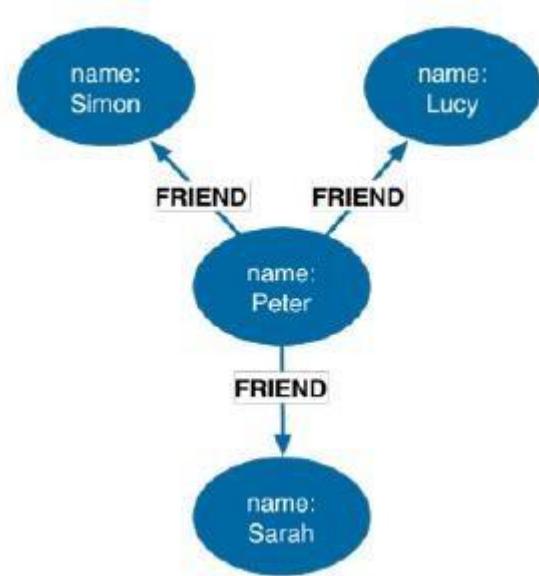
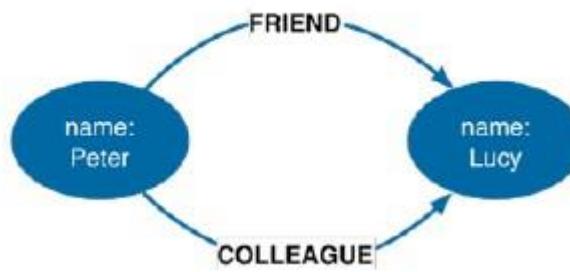
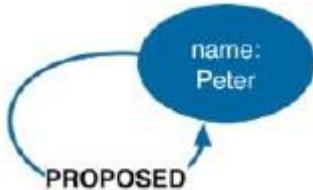


Edges

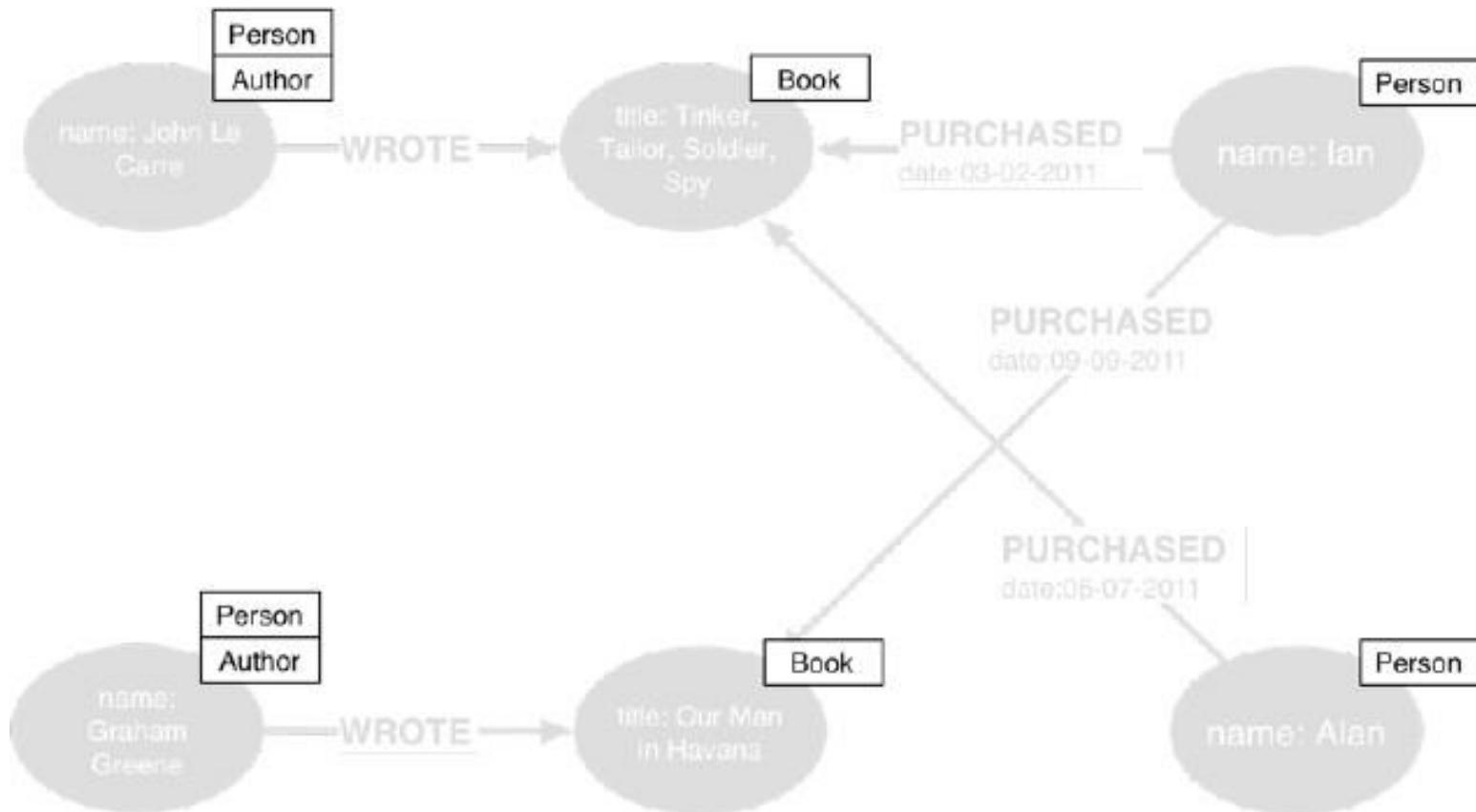


Edges

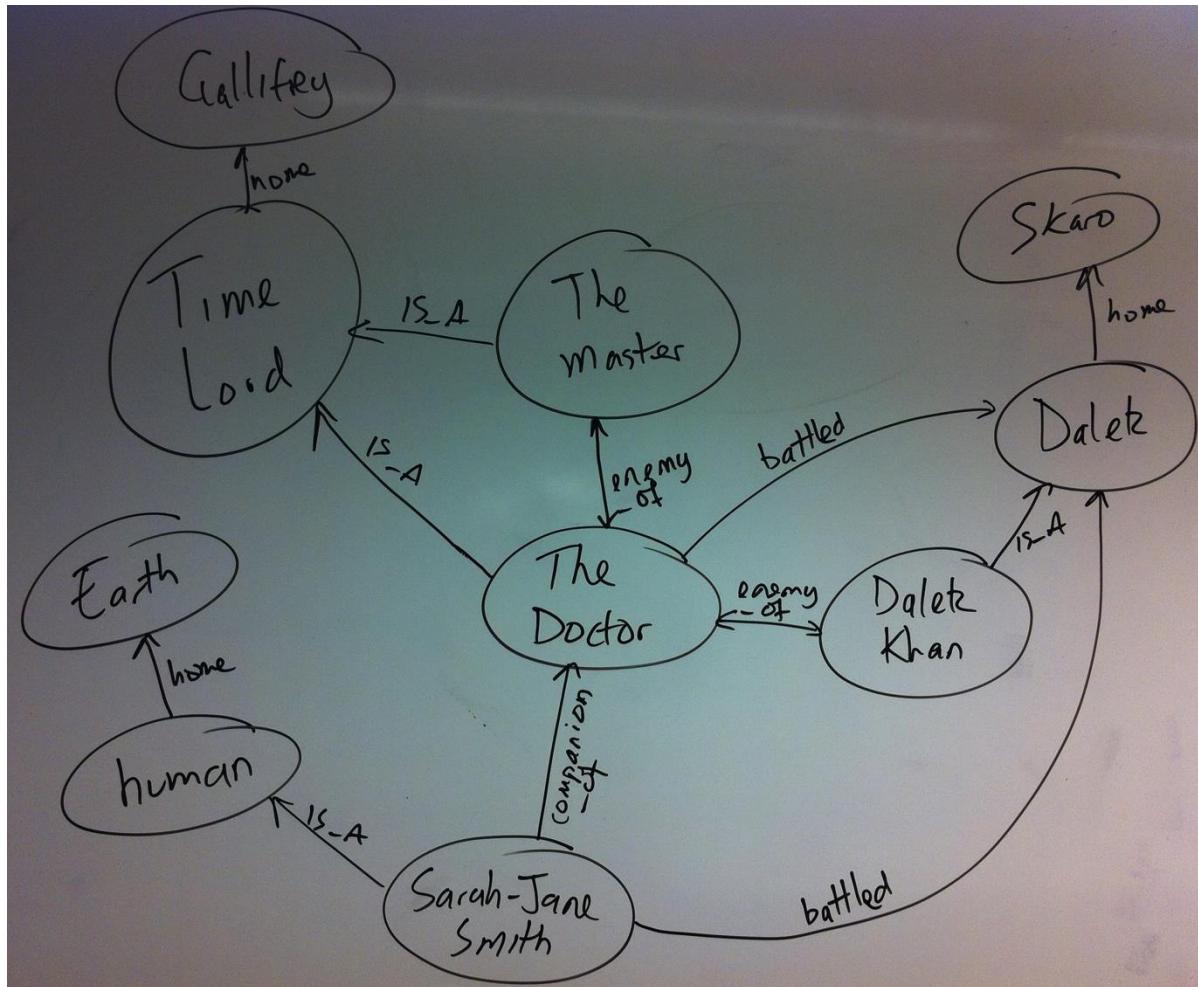
- One node can have multiple edges
- Can be
 - Parallel edges
 - loops



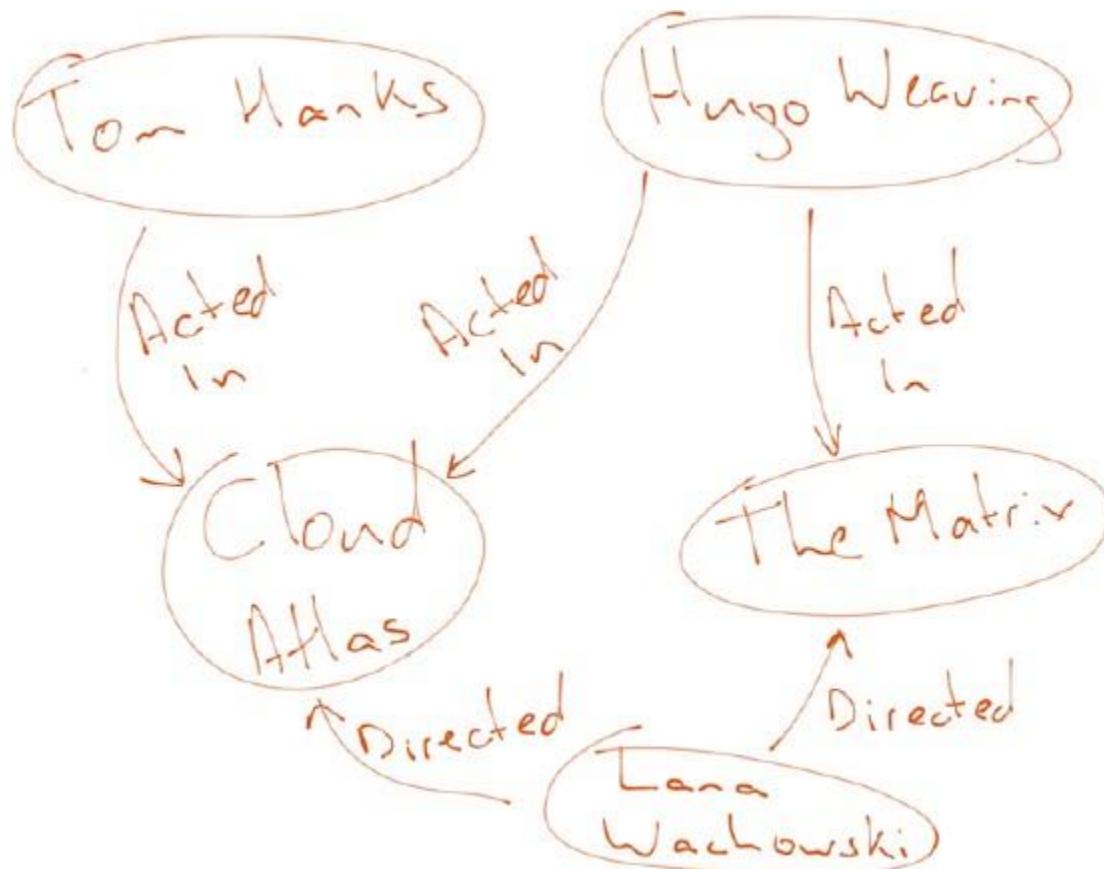
Labels



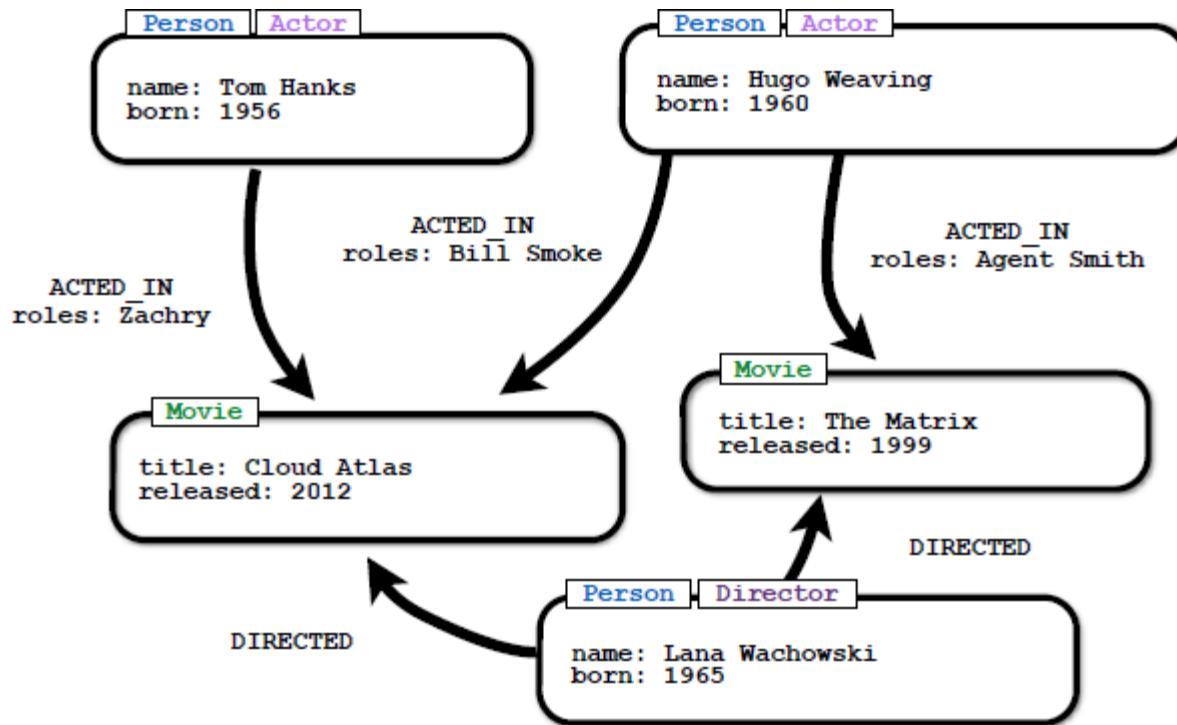
Graphs can be represented easily



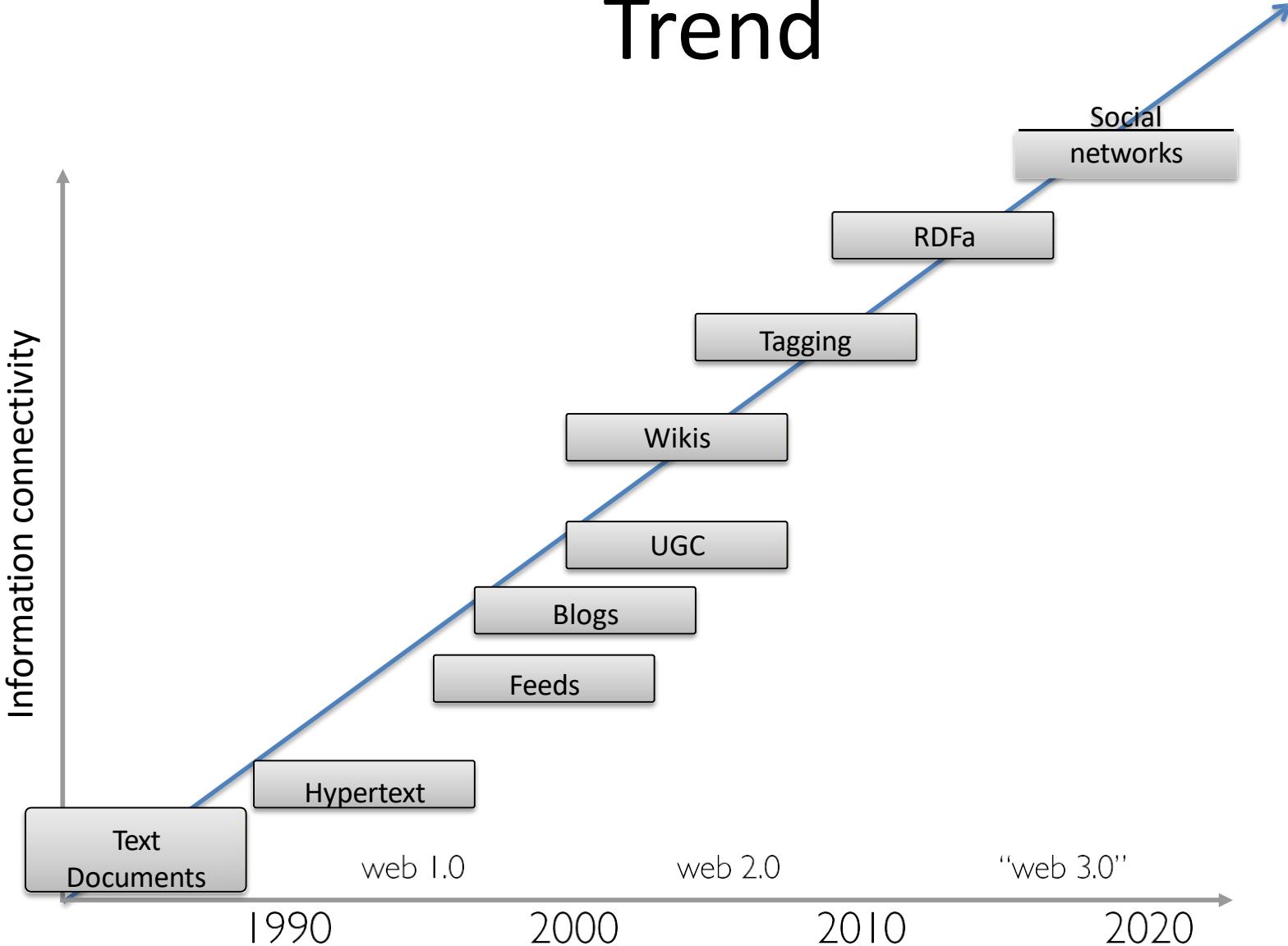
Simple model planning



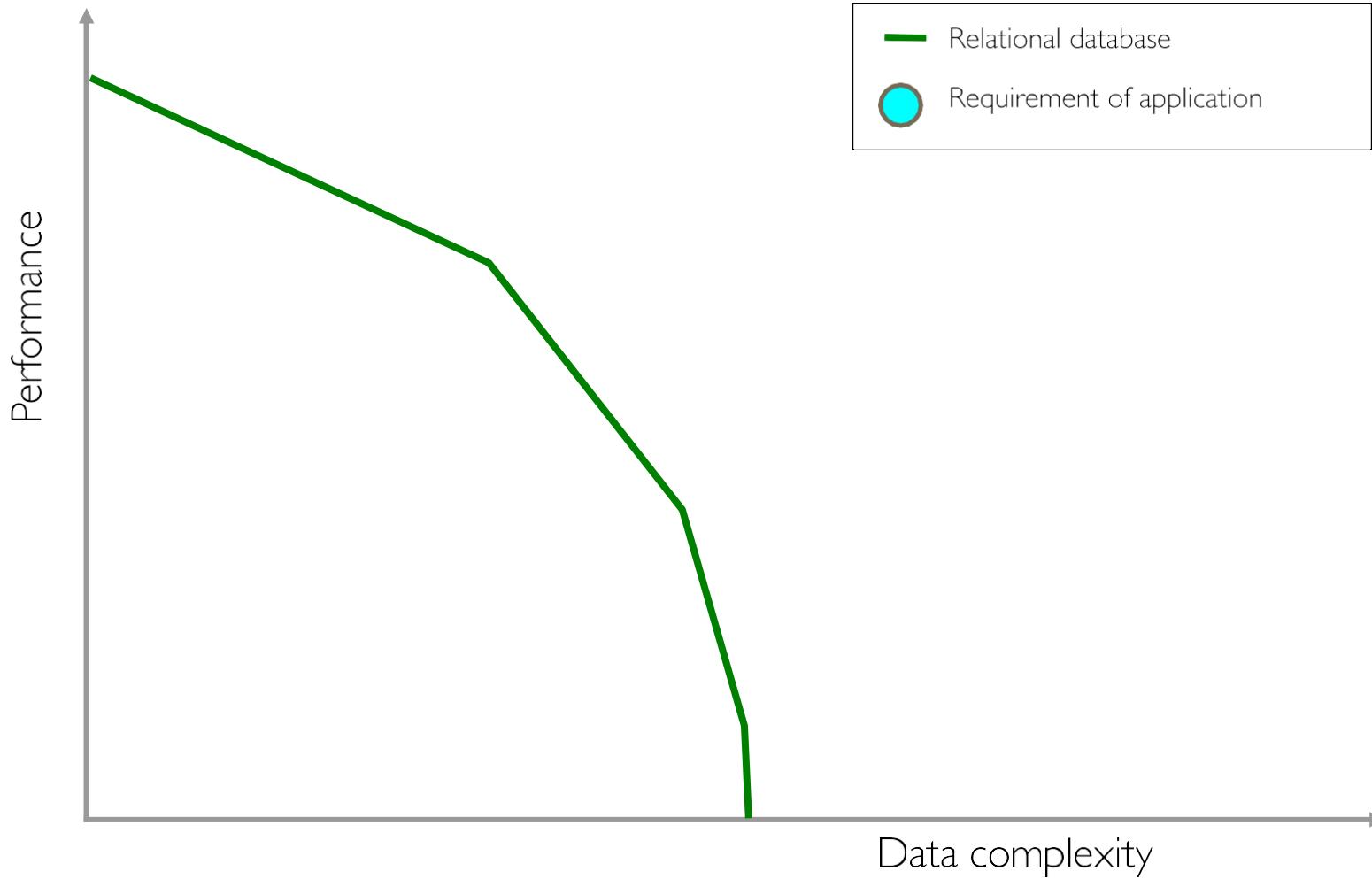
Simple model planning



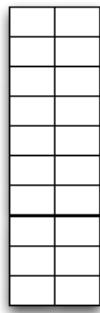
Trend



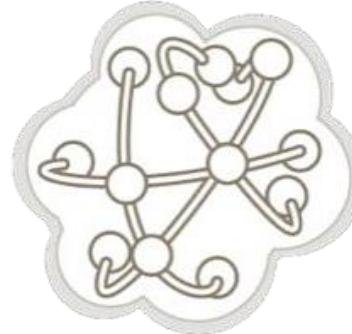
RDBMS performance



Key-Value



Graph DB

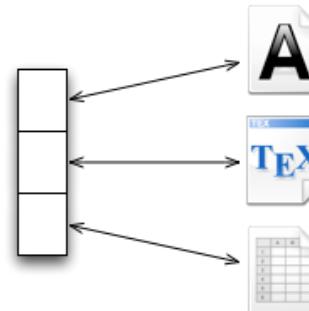


Four category of NOSQL

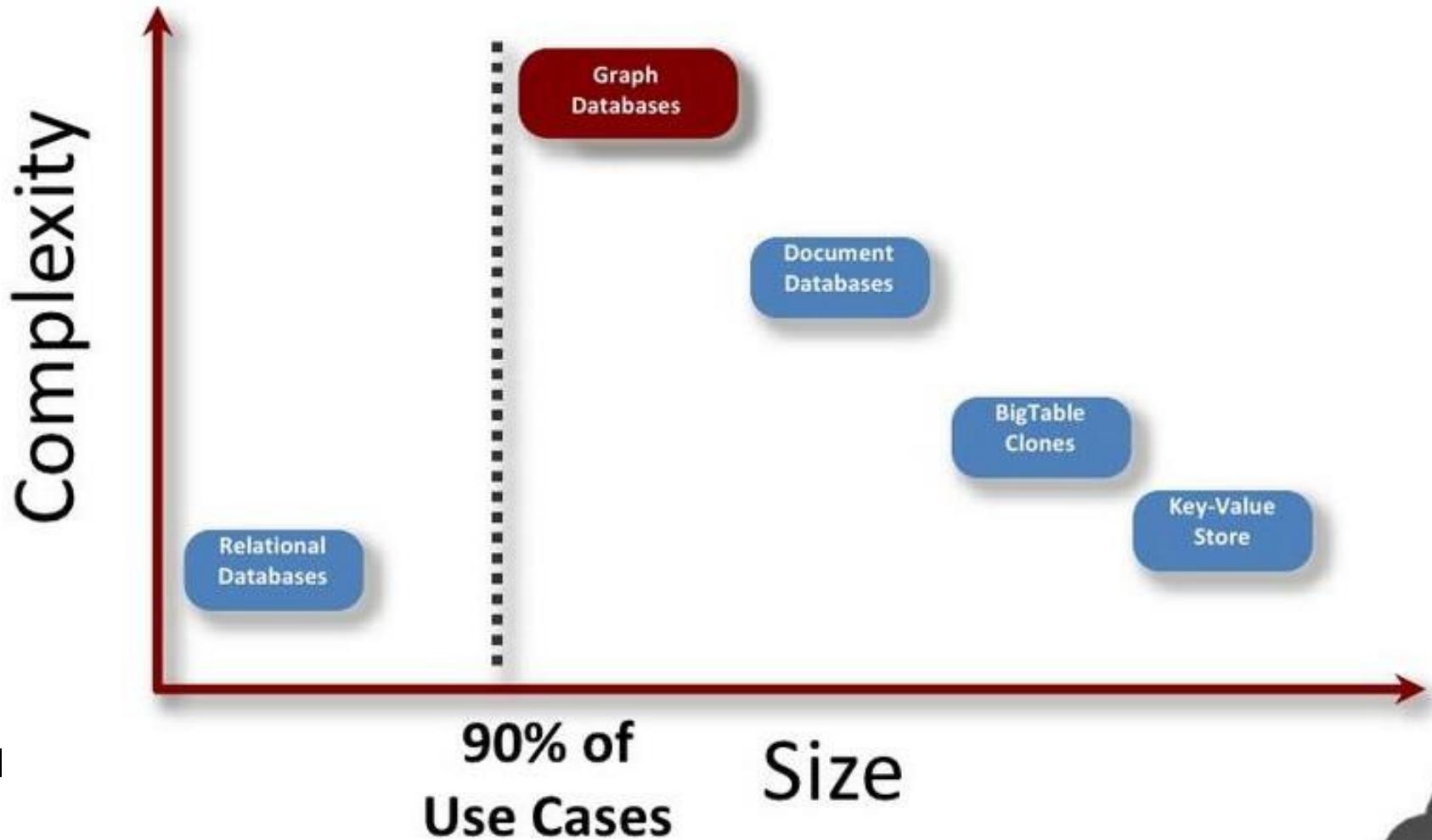
BigTable

1				1
	1		1	
	1	1		
				1
	1			1
	1			1
		1	1	
				1

Document



Living in a NOSQL World



Who uses?

- Facebook
 - Graph searches, searching common topics
- Google
 - Knowledge graph, for search
- Twitter
 - User recommendation
- Also
 - Recommendation, logistics, etc.

RDBMS vs. Graph databases

- Relational databases
 - Relationships between the elements of tables: foreign-keys
 - Bounded schema
 - Connections are calculated when querying
 - Many-to-many connections: connecting tables
- Graph databases
 - Every node is explicitly connected with its neighbors

RDBMS vs. Graph

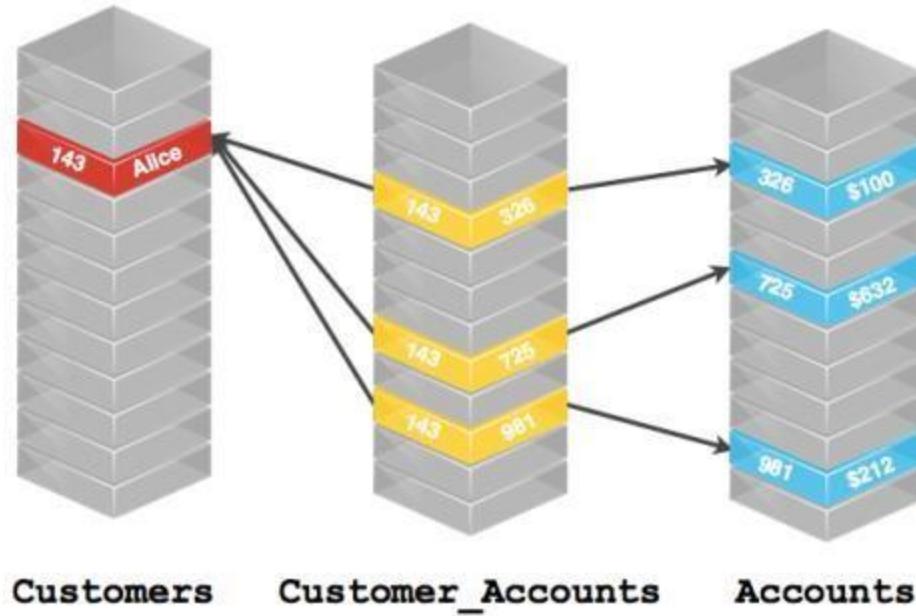


Customers

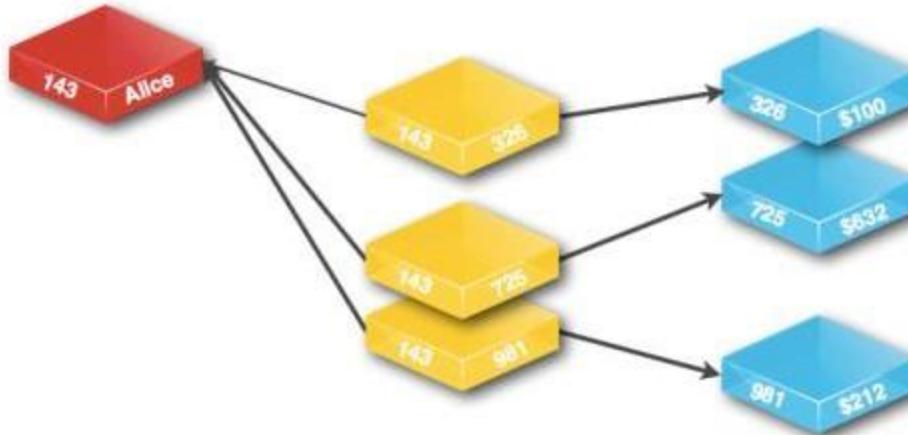


Accounts

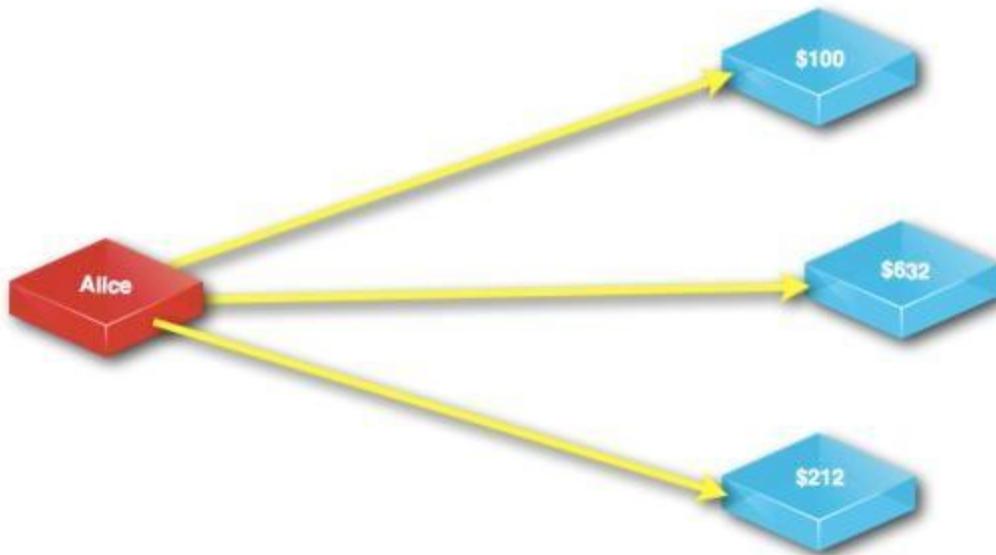
RDBMS vs Graph



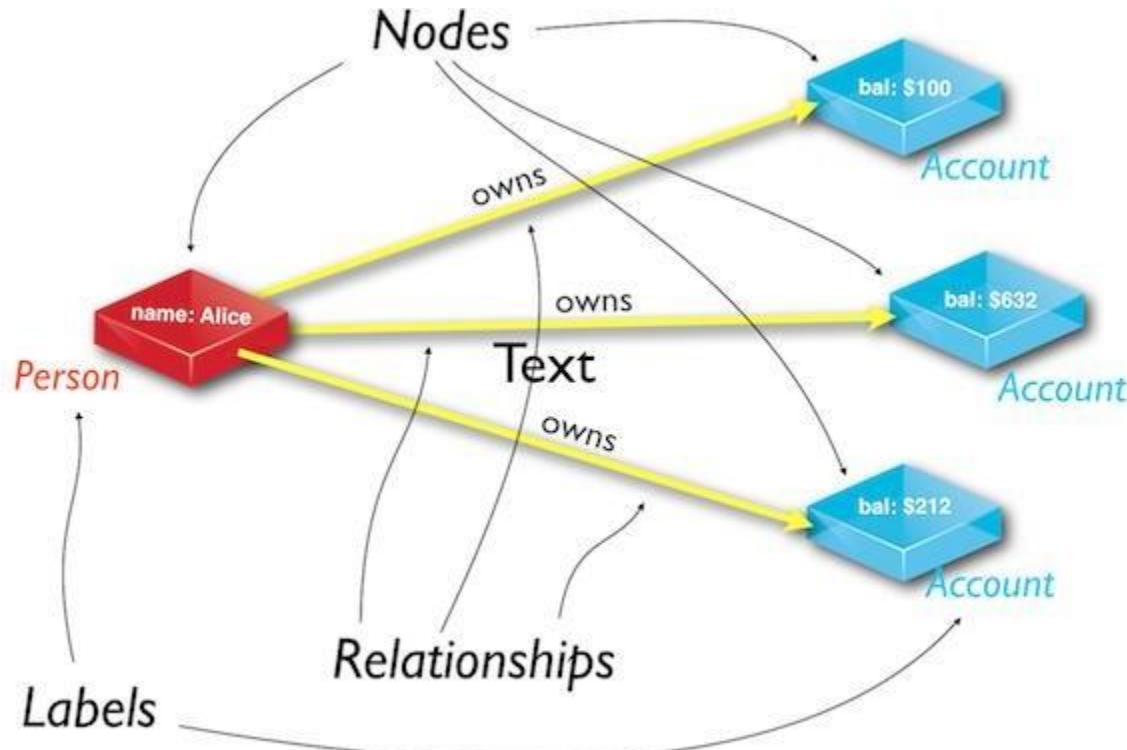
RDBMS vs Graph



RDBMS vs Graph



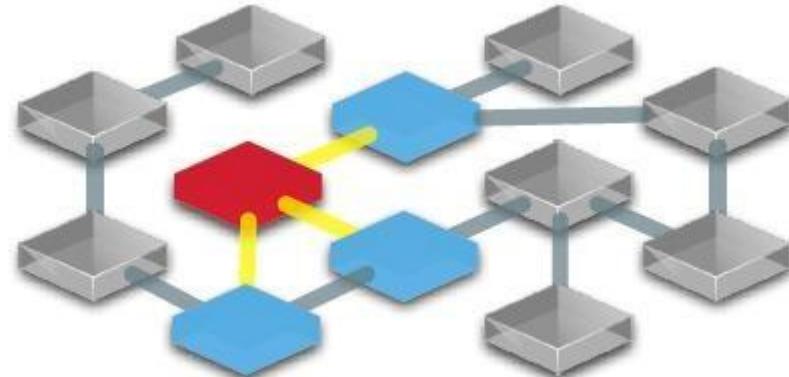
RDBMS vs Graph



© All Rights Reserved 2013 | Neo Technology, Inc.

RDBMS vs Graph

- Further relationships create greater graphs.
- Further people.
- Further Accounts.



Social networks „relationship” performance

- Measurement:

- ~1000 people
- Average 50 friend per person
- `pathExists(a, b)`
path length limit 4

	# people	runtime
RDBMS	1000	2000ms

Social networks „relationship” performance

- Measurement:

- ~1000 people
- Average 50 friend per person
- `pathExists(a, b)`
path length limit 4

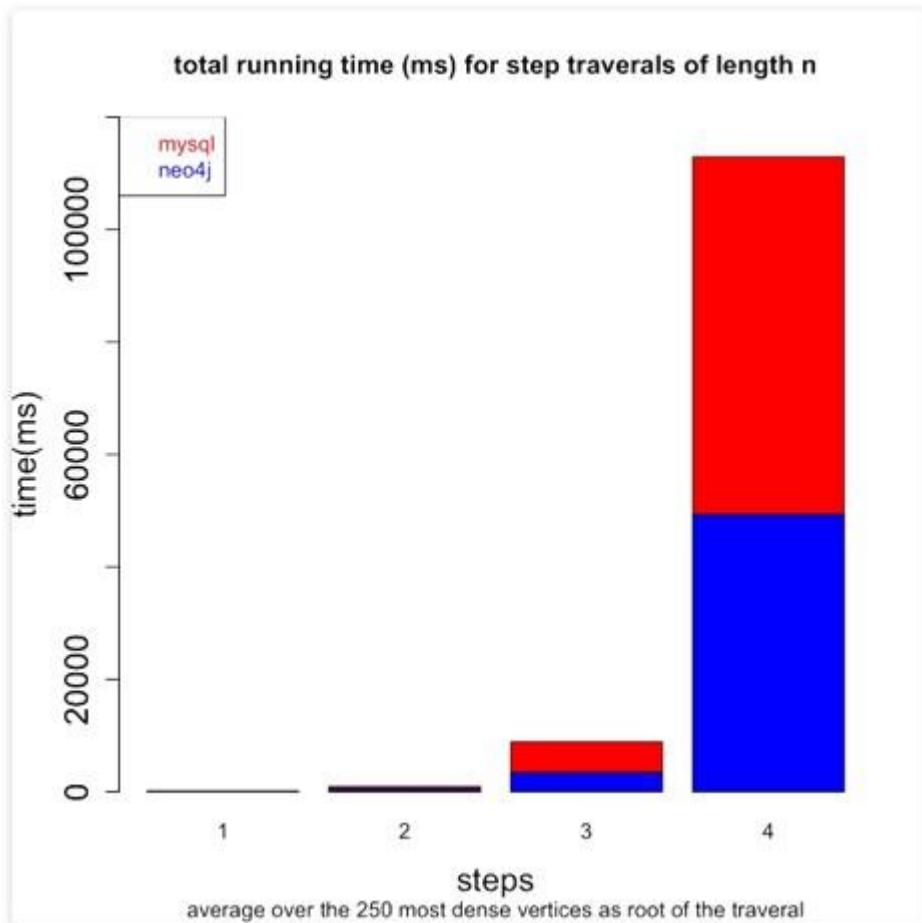
	# people	runtime
RDBMS	1000	2000ms
Neo4j	1000	2ms

Social networks „relationship” performance

- Measurement:
 - ~1000 people
 - Average 50 friend per person
 - `pathExists(a, b)`
path length limit 4

	# people	runtime
RDBMS	1000	2000ms
Neo4j	1000	2ms
Neo4j	1000000	2ms

MySQL vs Neo4j



```
CREATE TABLE graph (
    outV INT NOT NULL,
    inV INT NOT NULL
);
CREATE INDEX outV_index USING BTREE ON graph (outV);
CREATE INDEX inV_index USING BTREE ON graph (inV);
```

```
SELECT a.inV
FROM graph as a
WHERE a.outV=?
```

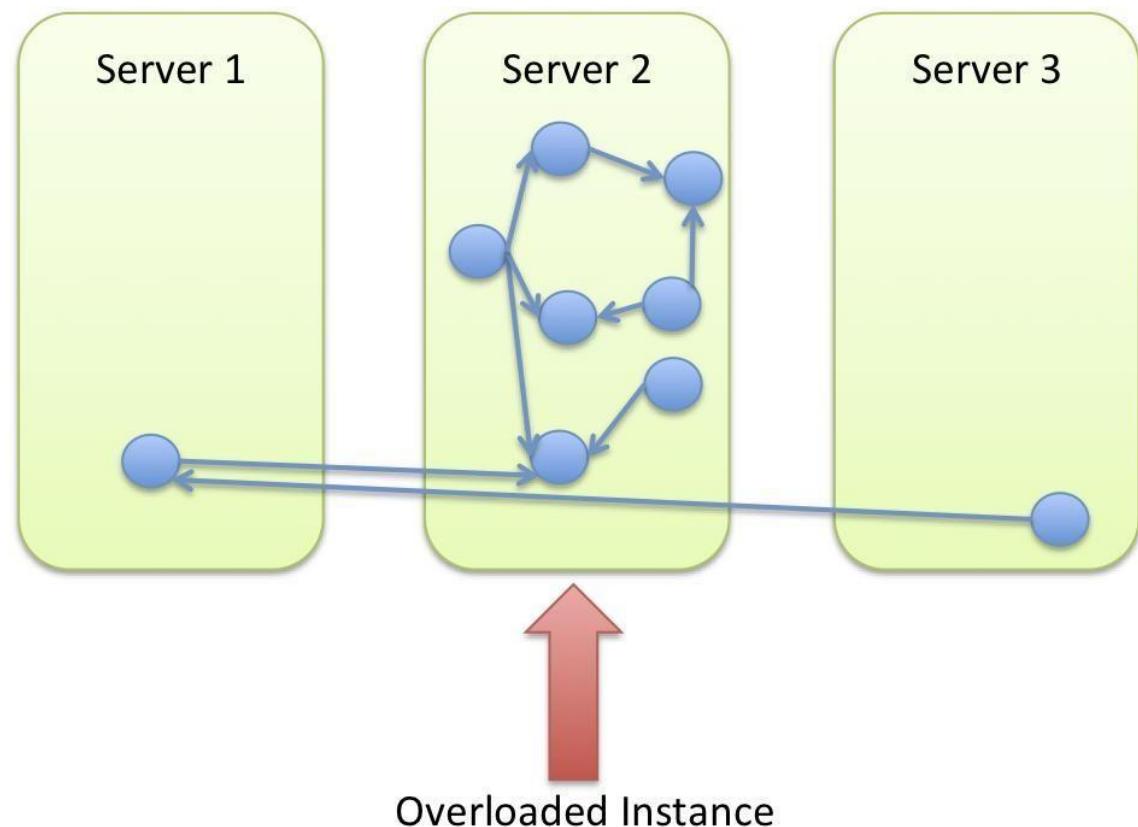
```
SELECT b.inV
FROM graph as a, graph as b
WHERE a.inV=b.outV and a.outV=?
```

Graph databases in cluster

- NoSql databases are running on clusters
- Cluster:
 - Connected group of computers
- A cluster can be scaled with new nodes
- How do we store a graph in cluster?

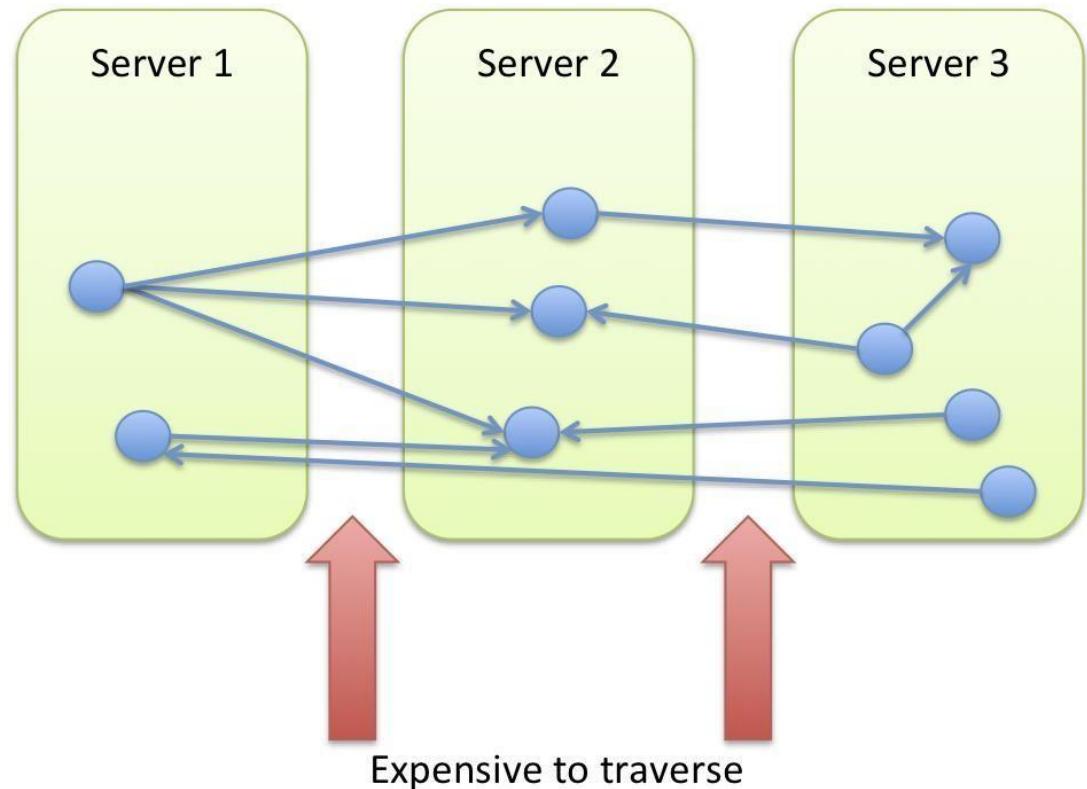
“Black hole” server

- Advantage:
 - Small communication between nodes
- Drawback:
 - Overloaded server



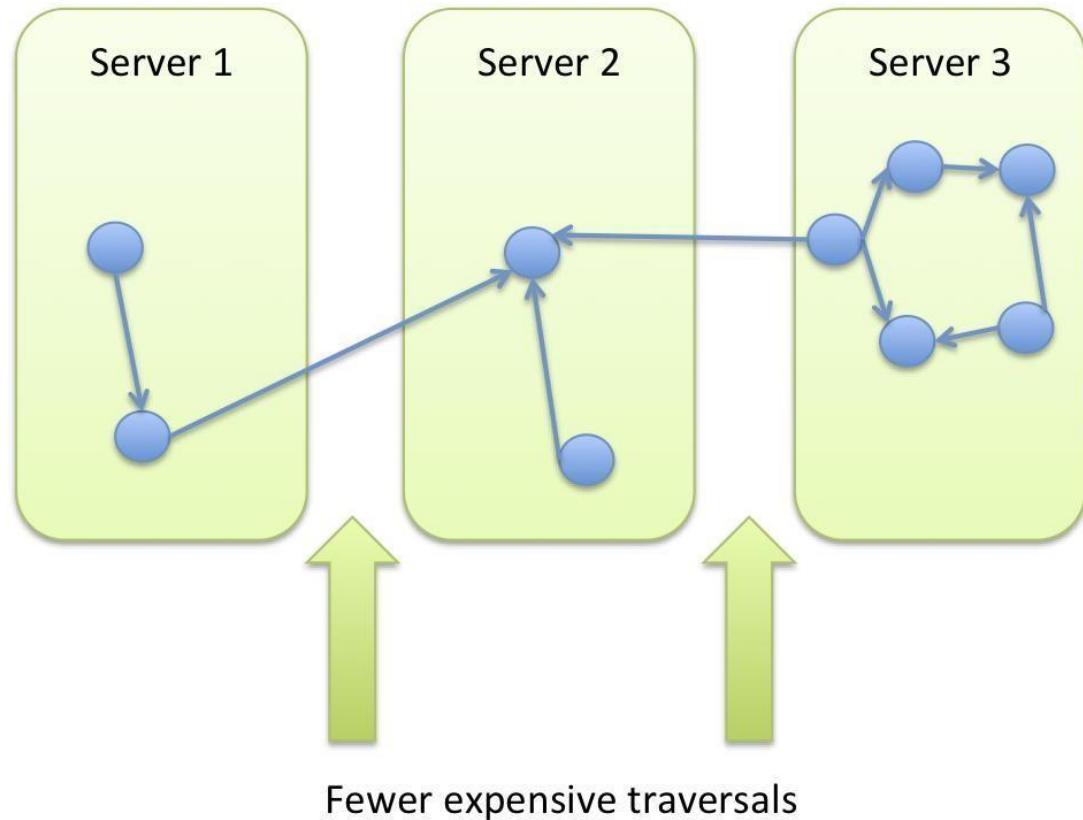
Chating network

- Advantage:
 - Steady dispersion of load
- Drawback:
 - Many communication between nodes



Minimal cut

- Best solution:
 - Load dispersion
 - Less communication between nodes



Neo4j

- Graph database
 - Schema independent
 - ACID transaction modell
 - Scalable
 - Java, embeddable
 - High availability (commercial)
 - Declarative query language
 - REST API



Graph databases: Advantages and drawbacks

- Advantage
 - Strong data modeling, more general than RDBMS
 - Faster query on joint data than in RDBMS
 - Easy to query
- Drawback:
 - Thinking in graphs
 - Global querying / calculation on node level
 - Storing binary data

Simple relationships

- **Nodes**
 - (a) actors
 - (m) movies
 - () anonymous node
- **Relationships**
 - [r]-> "r,, relationship
actors are in "r"
 - (a)-[r]->(m) relationship with movies
 - [:ACTED_IN]-> Type ACTED_IN relationship
(a)-[:ACTED_IN]->(m) actors are in ACTED_IN
(d)-[:DIRECTED]->(m) relationship movies
Directors are in DIRECTED
relationship with movies

Simple relationships

Connected
Nodes

Labels

Connected
Nodes



(a)-->()



(a:Person)



(a)-->(b)

Relationships

Relationships
Have Properties



(a)-[r]->()



(a)-[:ACTED_IN]->(m)

Properties

- For Nodes, relationships can have properties. Properties are Key-value pairs.
 - eg actor, name
- **Node attributes**

(m {title:"The Matrix"}) with Movie title property
(a {name:"Keanu Reeves",born:1964})
- **Relationship with Properties**

(a)-[:ACTED_IN {roles:["Neo"]}]>(m)
Relationship with (ACTED_IN) *roles* property

Labels

- Labels can help distinguish the nodes. A node can have multiple labels.
 - PI: *Clint Eastwood: Actor, Director.*
- **Labels**
 - (a:Person)
 - (a:Person {name:"Keanu Reeves"})
 - (a:Person)-[:ACTED_IN]->(m:Movie)

Cypher

- Cypher is the query language of Neo4j
- Cypher
 - declarative
 - Inspired by sql

Cypher

Query

- **MATCH:** Searching for give graph sample in graph.
- WHERE:** Filtering query.
- RETURN:** Resulting value, projection, aggregation.
- ORDER BY:** Ordering the result.
- SKIP/LIMIT:** Paging.

Cypher

Update

- **CREATE**: Creating node and edge.
- MERGE**: Creating unique node.
- CREATE UNIQUE**: Creating unique relationship.
- DELETE**: Deleting nodes, relationship.
- SET**: Setting up labels and properties.
- REMOVE**: Removing labels and properties.
- FOREACH**: Modifying elements in a given list.
- WITH**: Partitionating the query to different parts and giving the result to the next query.

Cypher

Two Nodes, One Relationship



```
MATCH (a)-->(b)  
RETURN a, b;
```

Returning a Property



```
MATCH (a)-->()  
RETURN a.name;
```

Naming a Relationship



```
MATCH (a)-[r]->()  
RETURN a.name, type(r);
```

Matching by Relationship

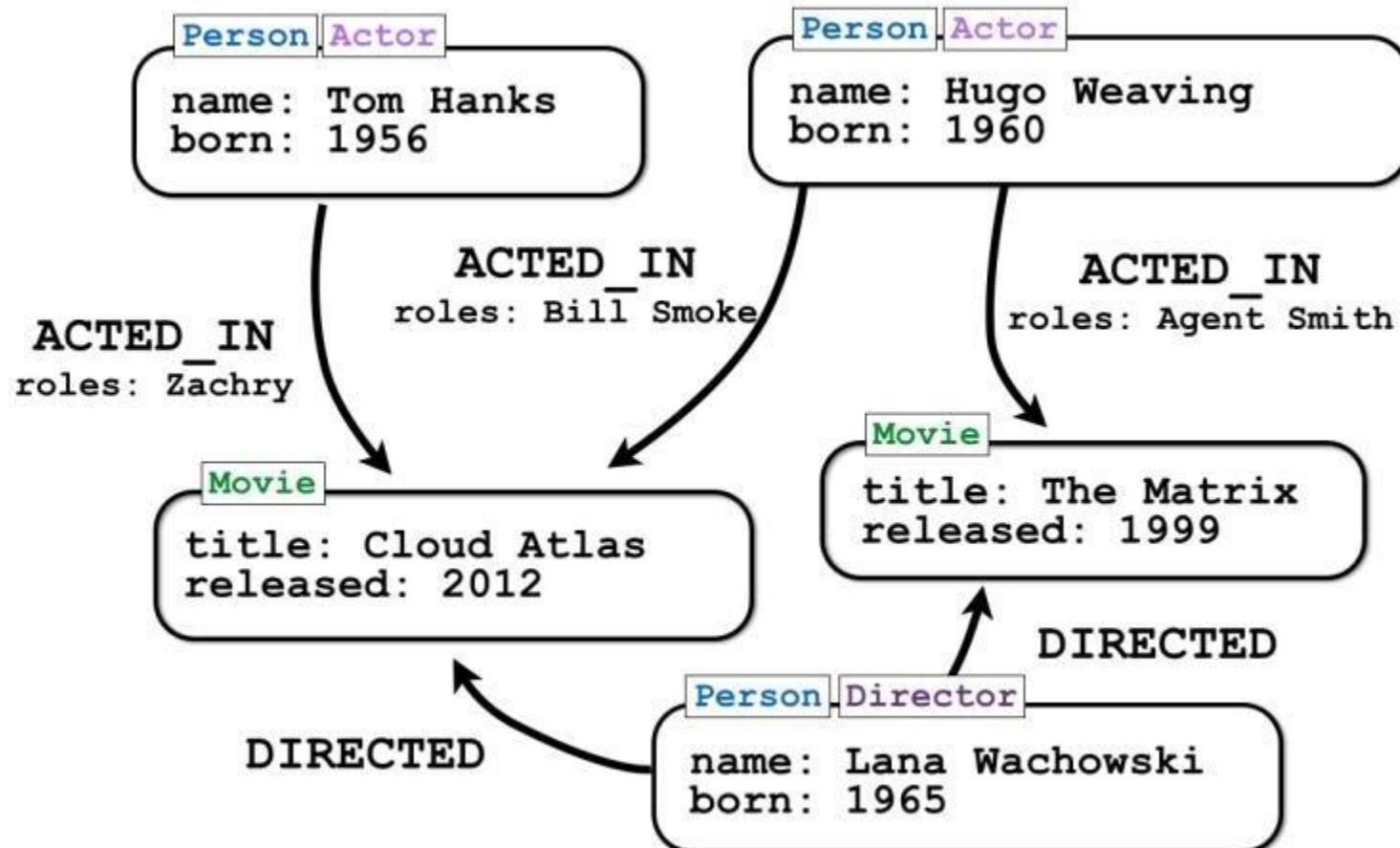


```
MATCH(a)-[:ACTED_IN]->(m)  
RETURN a.name, m.title;
```

Match Node Label

```
MATCH (tom:Person)  
WHERE tom.name="Tom Hanks"  
RETURN tom;
```

Example graph

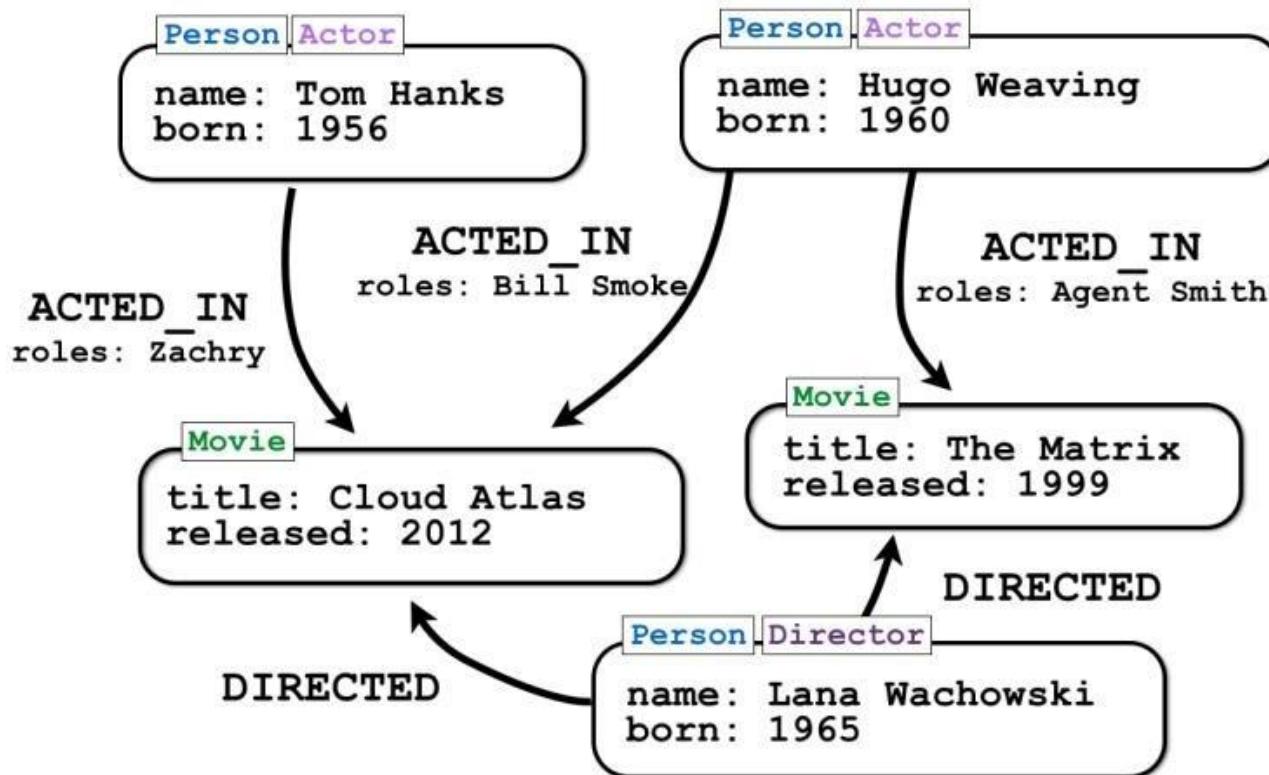


Querying all „Matrix” character

```
MATCH (m:Movie)<-[r:ACTED_IN]-(a:Person)
```

```
WHERE m.title=„The Matrix”
```

```
RETURN r.roles, a.name
```



Paths

- Multiple paths can be written together

```
MATCH (a)-[:ACTED_IN]->(m)<-[DIRECTED]-(d)  
RETURN a.name, m.title, d.name
```

- If there are too many, it is unreadable

```
MATCH (a)-[:ACTED_IN]->(m),  
      (m)<-[DIRECTED]-(d)  
RETURN a.name, m.title, d.name
```

Paths

- Paths can be named and results can be returned

```
MATCH p=(a)-[:ACTED_IN]->(m)<-[DIRECTED]-
(director)
RETURN p;
```

- And the nodes on the path

```
MATCH p=(a)-[:ACTED_IN]->(m)<-[DIRECTED]-
(director)
RETURN nodes(p);
```

Simplifying

- Example query

```
MATCH (p:Person)
WHERE p.name=„Tom Hanks”
RETURN p;
```

- Simplified

```
MATCH (p:Person {name=„Tom Hanks”})
RETURN p;
```

Limit, Skip, Distinct, Order by

- Skip: skipping results

```
MATCH (a)-[:ACTED_IN]->(m)
RETURN a.name, m.title
SKIP 10
LIMIT 10;
```

- Ordering

```
MATCH (p:Person)
RETURN p.name
ORDER BY p.born
LIMIT 10;
```

Distinct

```
MATCH (a)-[:ACTED_IN]->()
RETURN distinct a
ORDER BY a.born
LIMIT 10;
```

Complex query, Index

- Complex query

```
MATCH (g:Person {name: „Gene Hackman”})-[:ACTED_IN]->(movie),  
(other)-[:ACTED_IN]->(movie),  
(robin:Person {name:"Robin Williams"})  
WHERE NOT (robin)-[:ACTED_IN]->(movie)  
RETURN DISTINCT other;
```

- Index

```
CREATE INDEX ON :Movie(title);
```

```
CREATE INDEX ON :Person(name);
```

Aggregation

- Aggregation
 - Count
 - Min, Max
 - Avg
- Collect: Collecting

```
MATCH (a:Person)-[:ACTED_IN]->(m)  
RETURN a.name, collect(m.title);
```

Graph modification

- Since it is schema independent, any attribute can be added to any node and any edge

```
MATCH (m:Movie)
WHERE m.title=„Mystic River”
SET m.tagline= „BLABLA”
RETURN m;
```

- Attributes can be modified

```
MATCH (m:Movie)
WHERE m.title=„Mystic River”
SET m.released= 2003
RETURN m;
```

Merge

- Logical or for edges

```
MATCH (a)-[:ACTED_IN|DIRECTED]->()-<-[ACTED_IN|DIRECTED]-(b)  
RETURN a,b;
```

- Unifying edges

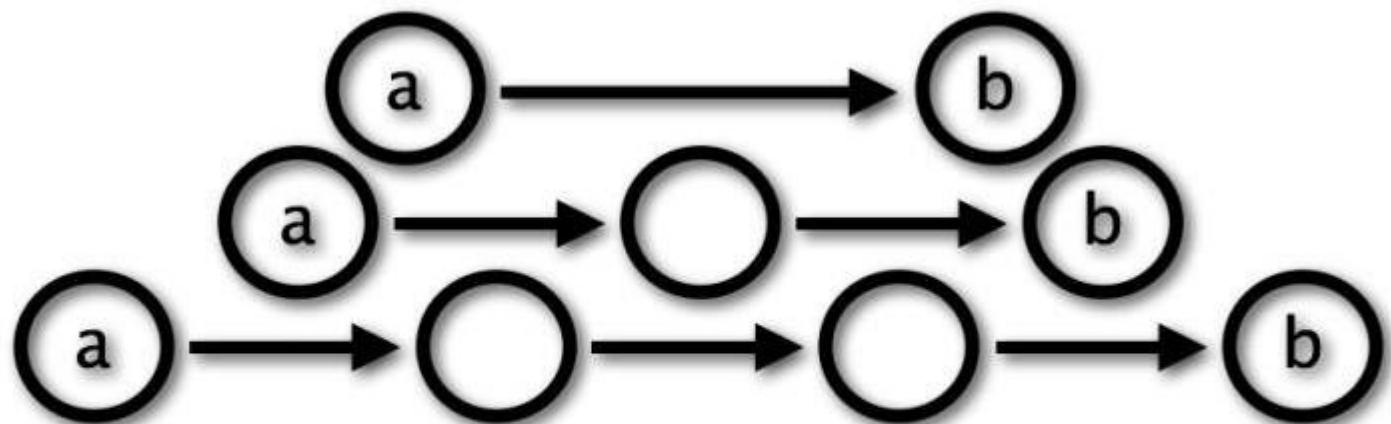
```
MATCH (a)-[:ACTED_IN|DIRECTED]->()-<-[ACTED_IN|DIRECTED]-(b)  
WHERE NOT a-[:KNOWS]-(b)  
MERGE (a)-[:KNOWS]->(b);
```

Optional

- Let's delete ourselves and our relationships

```
MATCH (me:Person {name=„My Name”})  
OPTIONAL MATCH (me)-[r]-()  
DELETE me, r;
```

Variable length paths



(a)-[*1..3]->(b)

shortestPath

- Shortest path function

```
MATCH p=shortestPath( (keanu:Person)-[:KNOWS*]->(kevin:Person) )
WHERE keanu.name=„Keanu Reeves” and kevin.name=„Kevin Bacon”
RETURN length(p);
```

```
MATCH (keanu:Person {name=„Keanu Reeves”}),
      (kevin:Person {name=„ Kevin Bacon”}),
MATCH p=shortestPath( (keanu)-[:KNOWS*]->(kevin) )
RETURN length(p);
```

Neo4j - original page 2 - neo4j-appendix

Overview

- Why do we use graphs?
- What is a graph?
- How do we work with graph databases (neo4j)

Programs: sones, neo4j, infinite graph(powered by Objectivity), Allegro Graph, dex, OrientDB, Hypergraphdb, rdf

Overview

- Why do we use graphs?
- What is a graph?
- How do we work with graph databases (neo4j)



NoSQL, Key-value databases, Redis

Péter Lehotay-Kéry
based on lecture notes of
Gergő Gombos

Redis

- REmote DIctionay Server
- „advanced, fast, persistent key-value database”
- Also called Data structure server

Redis

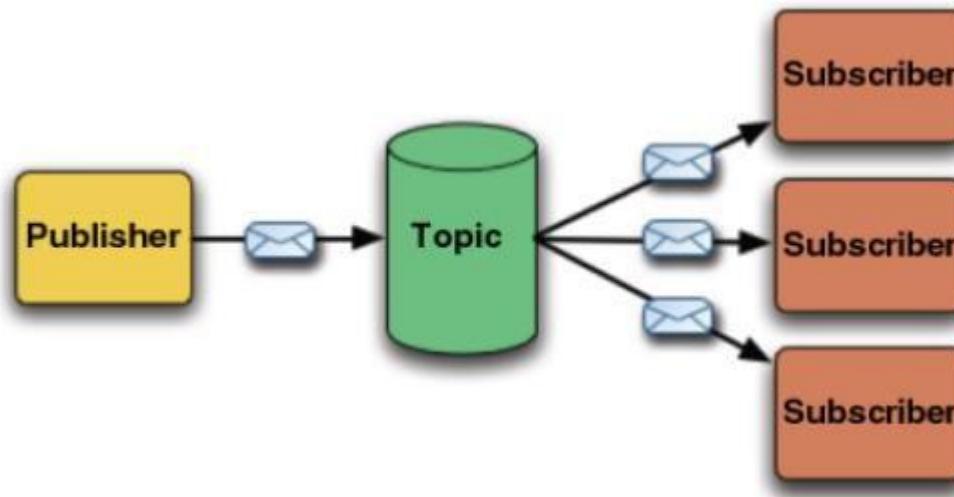
- Written in C
- In-memory
- Few dependencies, small size (1,3 MB)
- Fast
- One-thread
- Lots of client libraries
- First version: march 2009

Features

- Key-value database, where value:
 - string, list, set, sorted set, hash
- Replication: master-slave
- Publish – Subscribe
- Data with time limit
- Transactions

Publish/Subscribe

- Publisher sends data to the topic
- Clients subscribe to topics
- If new data arrives in the topic, clients get it



Persistence

- Snapshooting mode
 - Binary dumps, in every x seconds or after y operations
- Append Only File (AOF)
 - Every operation will be written out to a file
 - When restarting, every operation runs again
- Replication

Partitioning

- Redis cluster
 - Every node has to be available for the clients
 - Every node communicates with the others
 - Sharding with hash slots

Hash slots

- 16384 hash slot
- eg.:
 - A node : 0 5500
 - B node : 5501 11000
 - C node : 11001 16384
- Adding node
 - Copying some hash slot to the new node
- Removing node
 - Copying hash slots to the other nodes
- No need to stop cluster

Who uses?

- stackoverflow
- github
- blizzard
- twitter
- pinterest
- snapchat
- digg
- flickr
- vmware (sponsor)
- ...

Supported languages

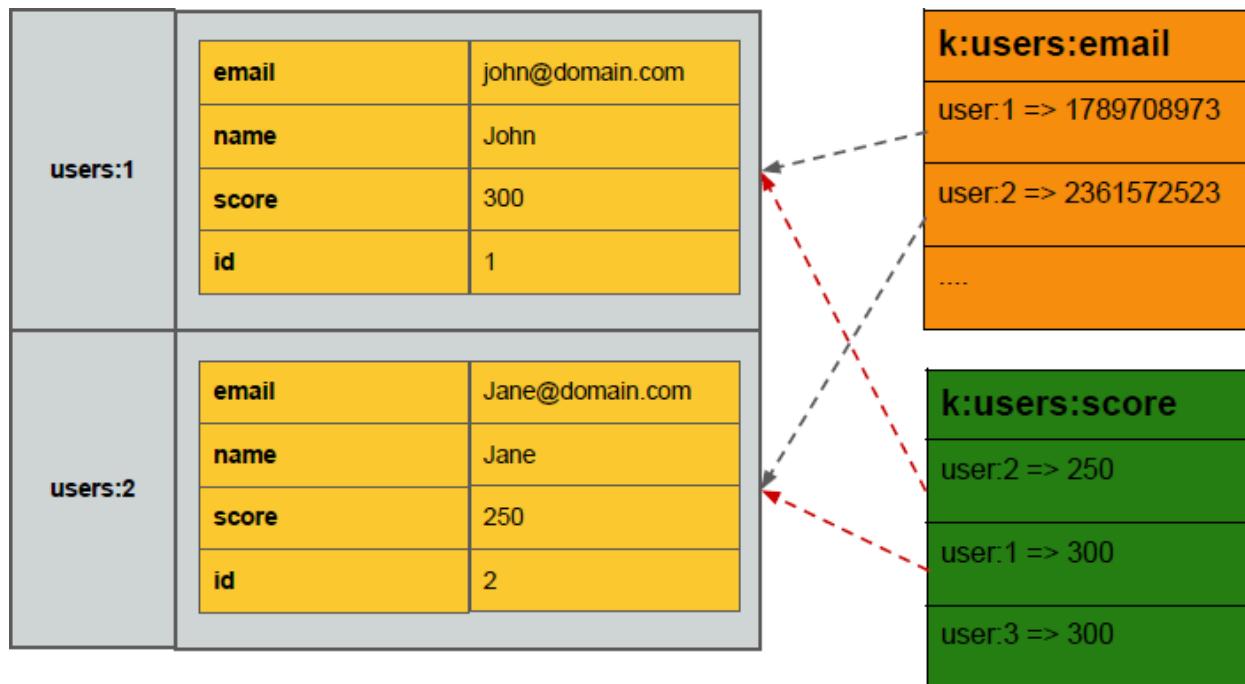
ActionScript	Bash	C	C#	C++	Clojure
Common Lisp	Crystal	D	Dart	Delphi	Elixir
emacs lisp	Erlang	Fancy	gawk	GNU Prolog	Go
Haskell	Haxe	Io	Java	Javascript	Julia
Lasso	Lua	Matlab	mruby	Nim	Node.js
Objective-C	OCaml	Pascal	Perl	PHP	Pure Data
Python	R	Racket	Rebol	Ruby	Rust
Scala	Scheme	Smalltalk	Swift	Tcl	VB
VCL					

Keys

- No long keys
- No whitespace, new line character
- „object-type-id:id:field” good, eg.:
„user:1001:name”

Indexes

- Manual indexes with sorted set
- Faster data reach of score and email (with hash)



Basic commands

- Querying variable type
 - TYPE mylist
- Querying help
 - HELP
- Redis informations
 - INFO

String value

- Setting value
 - SET mykey somevalue
- Querying value
 - GET mykey
- Deleting value
 - DEL mykey

String value

- Do not set if exists
 - SET mykey somevalue **nx**
 - SETNX mykey somevalue
- Set even if exists
 - SET mykey somevalue **xx**

Numeric values

- Setting
 - SET counter 100
- Increase
 - INCR counter
 - INCRBY counter 50
- Decrease
 - DECR counter
 - DECRBY counter 30

Why do we need function for increasing?

- We could solve it:
 - $x = \text{GET count}$
 - $x = x + 1$
 - $\text{SET count } x$
- Problem is:
 - A queries count, B also queries count
 - A increases count, B also increases count
 - What will be the result

Why do we need function to increase?

- We could solve it:
 - $x = \text{GET count}$
 - $x = x + 1$
 - $\text{SET count } x$
- Problem is:
 - A queries count(10), B also queries count (10)
 - A increases count (11), B also increases count (11)
 - Correct: 12

Key value

- Setting more values at once
 - mset a 10 b 20 c 30
- Getting multiple values
 - mget a b c

Time limit

- Setting value
 - SET key some-value
- Setting time
 - EXPIRE key 5
- Or at once
 - SET key 100 ex 10
- Querying time limit
 - TTL key

List

- Push value to list
 - RPUSH mylist A
 - LPUSH mylist first
 - RPUSH mylist 1 2 3 4 „foo”
- Query
 - LRANGE mylist 0 -1

List

- Poping data from list
 - RPOP mylist
 - LPOP mylist
- Blocked read from the list
 - BRPOP mylist 5 //waits 5 secs, if there's no element inside
- Keeping the last X field from the list
 - Useful for limits
 - LTRIM 0 2

Hash

- Setting value
 - HSET map key value
 - HMSET user:1000 username antirez birthyear 1977 verified 1
- Querying value
 - HGET user:1000 username
 - HGET user:1000 username birthday
 - HGETALL user:1000

Set

- Used for connections between objects
- Setting value
 - SADD myset 1 2 3
- Querying
 - SMEMBERS myset
- Is the value member of the set
 - SISMEMBER myset 3

Set

- SET example: news and labels
 - Connecting tags to news
 - SADD news:1000:tags 1 2
 - Connecting news to tags
 - SADD tag:1:news 1000
 - SADD tag:5:news 1000
 - SMEMBERS news:1000:tags
 - SMEMBERS tag:1:news

Set

- Union of sets
 - SUNIONSTORE newSet Set1 Set2
- Copying set
 - SUNIONSTORE newSet Set1
- Extract element from set – random
 - SPOP newSet
- Count of set
 - SCARD newSet

Sorted Set

- Set with score value
- pl.: ZADD hackers 1912 „Alan Turing“
ZADD hackers 1965 "Yukihiro Matsumoto,"
- Score is birth time.

Sorted Set

- Query
 - ZRANGE hackers 2 4 //query 2-4 elements
 - ZRANGE hackers 0 -1 //all elements from 0
 - ZREVRANGE hackers 0 -1 //in decreasing order
 - ZRANGE hackers 0 -1 withscore //write out score
 - ZRANGEBYSCORE hackers –inf +inf //between infinities
 - ZRANGEBYSCORE hackers (1 2 //1 < <= 2
 - ZREMRANGEBYSCORE hackers 1 3 //eliminating values between 1 and 3

Handling transaction

- Use transaction to execute more commands
 - MULTI //start of transaction
 - //commands
 - EXEC //executing transaction
- Not reading value, while not running

Consistency of distributed systems

- We store data on multiple machines
- Machines can malfunction
- We can reach the data through multiple access points
 - Access should not be bottleneck
- Want to ensure:
 - Consistency
 - Availability
 - Partition tolerance

Consistency (C)

- Replication instead of safety saves
 - Data availability at multiple places
- What ensures that replicates stay consistent?
 - Need some protocol
 - Usually asynchronous
- Consistence window
 - After how much time the system will become consistent?

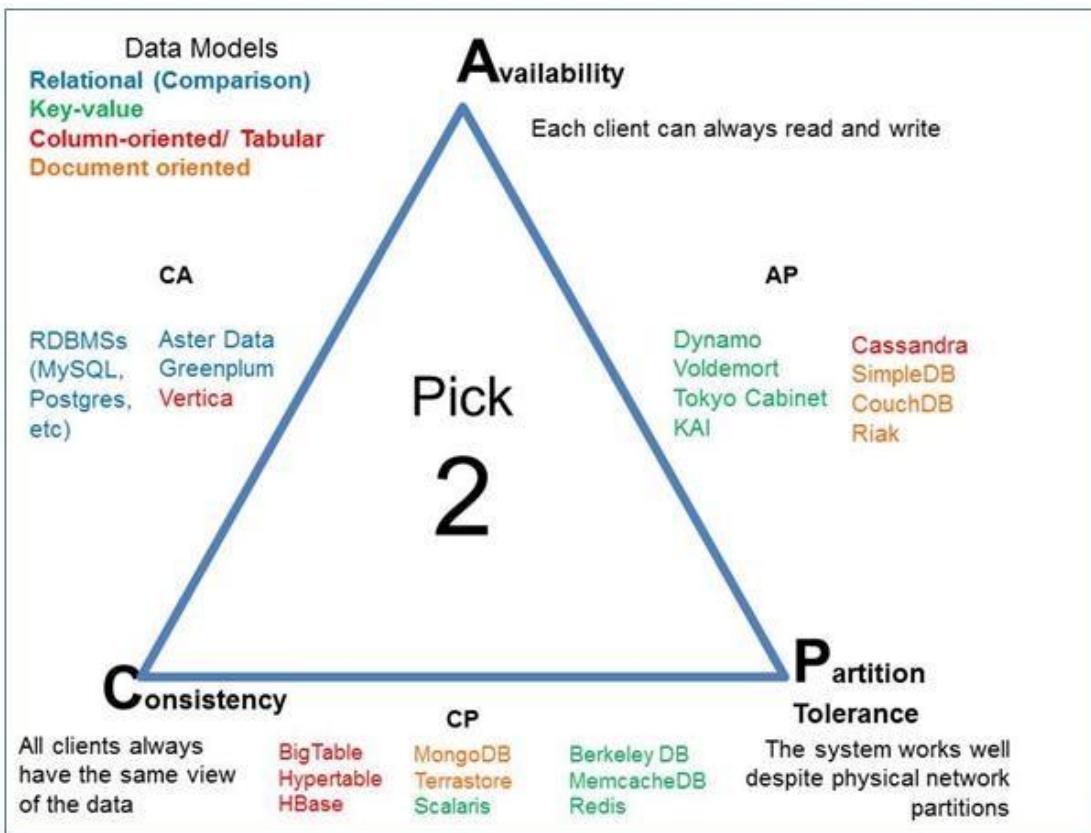
Availability (A)

- Data can always be reached
 - Multiple access points
 - No critical elements
 - Geo-redundancy
- Quick answers
 - Minimal delay
 - Even if the returned data is inconsistent
(eg Facebook timeline)

Partition tolerance (P)

- Distributed system
 - Network connection (slow, fragile)
- Multiple access points
 - System is functional if individual parts cannot see each other
- Distributed functionality
 - Eg Facebook user login, sharing pictures, message box

CAP theorem



ACID transaction model

- Atomicity
 - Transaction fully executed, or not executed at all
- Consistency
 - As the effect of transaction, database comes from consistent state arrives to consistent state
- Isolation
 - Concurrent execution of transactions must be equivalent to a sequential execution result
- Durability
 - If a transaction is successful, the change must be durable

BASE transaction model

- BA: Basically available
 - At least one part of the system remains reachable
- Soft-state
 - Changes happen through finite time messages
 - State of system can change even if there is no input
 - Any data may have validity time
 - If it runs out, it must be examined, if it is still consistent
- Eventually consistent
 - Changes propagate asynchronous
 - Mostly in AP systems
 - Changes become consistent after a time
 - Consistency window

Resolving inconsistency

- Errors can cause inconsistency
 - Must be resolved: when reading, writing, asynchronous
- Algorithms based on consensus
 - Nodes containing replicates vote
- Solutions based on time stamps
 - We always consider the latest data

1. 1. What's in it for you? What is HBase? 1
2. 2. What's in it for you? What is HBase? 1 HBase Use Case 2
3. 3. What's in it for you? What is HBase? Applications of HBase 1 HBase Use Case 2 3
4. 4. What's in it for you? What is HBase? Applications of HBase HBase vs RDBMS 1 HBase Use Case 2 3 4
5. 5. What's in it for you? What is HBase? Applications of HBase HBase Storage HBase vs RDBMS 1 HBase Use Case 2 3 4 5
6. 6. What's in it for you? What is HBase? Applications of HBase HBase Storage HBase Architectural Components HBase vs RDBMS 1 HBase Use Case 2 3 4 5 6
7. 7. What's in it for you? What is HBase? Applications of HBase HBase Storage HBase Architectural Components Demo on HBase HBase vs RDBMS 1 HBase Use Case 2 3 4 5 6 7
8. 8. Introduction to HBaseIntroduction to HBase
9. 9. Introduction to HBase This data could be easily stored in a Relational Database (RDMS) Structured data Back in the days, data used to be less and was mostly structured
10. 10. Introduction to HBase Then, Internet evolved and huge volumes of structured and semi-structured data got generated Storing and processing this data on RDBMS became a major problem Semi-structured data
11. 11. Introduction to HBase Apache HBASE was the solution for this Semi-structured data SolutionThen, Internet evolved and huge volumes of structured and semi- structured data got generated
12. 12. Introduction to HBaseHBase History
13. 13. HBase History 1 Google released the paper on BigTable Nov 2006
14. 14. HBase History 1 2 Google released the paper on BigTable HBase prototype was created as a Hadoop contribution Nov 2006 Feb 2007
15. 15. HBase History 1 2 3 Google released the paper on BigTable HBase prototype was created as a Hadoop contribution First usable HBase along with Hadoop 0.15.0 was released Nov 2006 Feb 2007 Oct 2007
16. 16. HBase History 1 2 3 4 Google released the paper on BigTable HBase prototype was created as a Hadoop contribution First usable HBase along with Hadoop 0.15.0 was released HBase became the subproject of Hadoop Nov 2006 Feb 2007 Oct 2007 Jan 2008
17. 17. HBase History 1 2 3 4 5 Google released the paper on BigTable HBase prototype was created as a Hadoop contribution First usable HBase along with Hadoop 0.15.0 was

released HBase became the subproject of Hadoop HBase 0.81.1, 0.19.0 and 0.20.0 was released between Oct 2008 – Sep 2009 Nov 2006 Feb 2007 Oct 2007 Jan 2008 Oct 2008 – Sep 2009

18. [18. HBase History](#) 1 2 3 4 5 6 Google released the paper on BigTable HBase prototype was created as a Hadoop contribution First usable HBase along with Hadoop 0.15.0 was released HBase became the subproject of Hadoop HBase 0.81.1, 0.19.0 and 0.20.0 was released between Oct 2008 – Sep 2009 HBase became Apache top-level project Nov 2006 Feb 2007 Oct 2007 Jan 2008 Oct 2008 – Sep 2009 May 2010
19. [19. Introduction to HBase](#)What is HBase?
20. [20. What is HBase?](#) HBase is a column oriented database management system derived from Google's NoSQL database BigTable that runs on top of HDFS
21. [21. What is HBase?](#) HBase is a column oriented database management system derived from Google's NoSQL database BigTable that runs on top of HDFS Open source project that is horizontally scalable¹
22. [22. What is HBase?](#) Open source project that is horizontally scalable¹ 2 HBase is a column oriented database management system derived from Google's NoSQL database BigTable that runs on top of HDFS NoSQL database written in JAVA which performs faster querying
23. [23. What is HBase?](#) Open source project that is horizontally scalable NoSQL database written in JAVA which performs faster querying Well suited for sparse data sets (can contain missing or NA values) 1 2 3 HBase is a column oriented database management system derived from Google's NoSQL database BigTable that runs on top of HDFS
24. [24. Introduction to HBase](#)Companies using HBase
25. [25. Companies using HBase](#)
26. [26. Introduction to HBase](#)HBase Use Case
27. [27. HBase Use Case](#) Telecommunication company that provides mobile voice and multimedia services across China
28. [28. HBase Use Case](#) Telecommunication company that provides mobile voice and multimedia services across China Generated billions of Call Detail Records (CDR)
29. [29. HBase Use Case](#) Telecommunication company that provides mobile voice and multimedia services across China Traditional database systems were unable to scale up to the vast volumes of data and provide a cost-effective solution Generated billions of Call Detail Records (CDR)

30. [30.](#) HBase Use Case Telecommunication company that provides mobile voice and multimedia services across China Generated billions of Call Detail Records (CDR) Traditional database systems were unable to scale up to the vast volumes of data and provide a cost-effective solution
31. [31.](#) HBase Use Case Telecommunication company that provides mobile voice and multimedia services across China Storing and real-time analysis of billions of call records was a major problem Generated billions of Call Detail Records (CDR)
32. [32.](#) HBase Use Case Telecommunication company that provides mobile voice and multimedia services across China HBase stores billions of rows of detailed call records Solution Generated billions of Call Detail Records (CDR)
33. [33.](#) HBase Use Case Telecommunication company that provides mobile voice and multimedia services across China HBase performs fast processing of records using SQL queries Generates billions of Call Detail Records (CDR)
34. [34.](#) Introduction to HBase Applications of HBase
35. [35.](#) Applications of HBase Medical HBase is used for storing genome sequences Storing disease history of people or an area
36. [36.](#) Applications of HBase Medical E-Commerce HBase is used for storing genome sequences Storing disease history of people or an area HBase is used for storing logs about customer search history Performs analytics and target advertisement for better business insights
37. [37.](#) Applications of HBase Medical E-Commerce Sports HBase is used for storing genome sequences Storing disease history of people or an area HBase is used for storing logs about customer search history Performs analytics and target advertisement for better business insights HBase stores match details and history of each match Uses this data for better prediction
38. [38.](#) Introduction to HBase HBase vs RDBMS
39. [39.](#) HBase vs RDBMS Does not have a fixed schema (schema-less). Defines only column families Has a fixed schema which describes the structure of the tables HBase RDBMS
40. [40.](#) HBase vs RDBMS Does not have a fixed schema (schema-less). Defines only column families Has a fixed schema which describes the structure of the tables Works well with structured and semi-structured data Works well with structured data HBase RDBMS

41. [41.](#) HBase vs RDBMS Does not have a fixed schema (schema-less). Defines only column families Has a fixed schema which describes the structure of the tables Works well with structured and semi-structured data Works well with structured data RDBMS can store only normalized data HBase RDBMS It can have de-normalized data (can contain missing or NA values)
42. [42.](#) HBase vs RDBMS Does not have a fixed schema (schema-less). Defines only column families Has a fixed schema which describes the structure of the tables Works well with structured and semi-structured data Works well with structured data It can have de-normalized data (can contain missing or NA values) RDBMS can store only normalized data Built for wide tables that can be scaled horizontally Built for thin tables that is hard to scale HBase RDBMS
43. [43.](#) Introduction to HBaseFeatures of HBase
44. [44.](#) Features of HBase Scalable Data can be scaled across various nodes as it is stored in HDFS
45. [45.](#) Features of HBase Scalable Data can be scaled across various nodes as it is stored in HDFS Automatic failure support Write Ahead Log across clusters which provides automatic support against failure
46. [46.](#) Features of HBase Scalable Data can be scaled across various nodes as it is stored in HDFS Consistent read and write HBase provides consistent read and write of data Automatic failure support Write Ahead Log across clusters which provides automatic support against failure
47. [47.](#) Features of HBase Scalable Data can be scaled across various nodes as it is stored in HDFS Consistent read and write HBase provides consistent read and write of data JAVA API for client access Provides easy to use JAVA API for clients Automatic failure support Write Ahead Log across clusters which provides automatic support against failure
48. [48.](#) Features of HBase Scalable Data can be scaled across various nodes as it is stored in HDFS Consistent read and write HBase provides consistent read and write of data JAVA API for client access Provides easy to use JAVA API for clients Automatic failure support Write Ahead Log across clusters which provides automatic support against failure Block cache and bloom filters Supports block cache and bloom filters for high volume query optimization
49. [49.](#) Introduction to HBaseHBase Storage

50. [50.](#) HBase column oriented storage Column Family 1 Column Family 2 Column Family 3
Rowid Col 1 Col 2 Row 1 Row 2 Row 3 Col 3 Col 3Col 1 Col 2 Col 3Col 1 Col 2 Row Key
Column Family Column Qualifiers Cells
51. [51.](#) HBase column oriented storage Personal data Professional data Rowid name 1 2 3 Row
Key Column Family Column Qualifiers Cells city age salary empid Angela Dwayne David
Chicago Boston Seattle 31 35 29 Data Analyst Web Developer Big Data Architect \$70,000
\$65,000 \$55,000 designation
52. [52.](#) Introduction to HBase HBase Architecture
53. [53.](#) HBase Architectural Components Region Server HLog MemStore StoreFile StoreFile
HFile StoreRegion Region Server HLog MemStore StoreFile StoreFile HFile HFile
StoreRegion Region Server HLog MemStore StoreFile StoreFile HFile HFile StoreRegion
HDFS HMaster HBase Master assigns regions and load balancing ZooKeeper is used for
monitoring Region server serves data for read and write
54. [54.](#) HBase Architectural Components - Regions Key col col xxx val val xxx val val Key col col
xxx val val xxx val val Key col col xxx val Val xxx val Val Key col col xxx val val xxx val val
Region 1 Region 2 Region 3 Region 4 startKey endKey endKey Client HBase tables
are divided horizontally by row key range into "Regions" A region contains all rows in the
table between the region's start key and end key Regions are assigned to the nodes in the
cluster, called "Region Servers" These servers serve data for read and write startKey get
Region Server 1 Region Server 2
55. [55.](#) HBase Architectural Components - HMaster Key col col xxx val val xxx val val Key col col
xxx val val xxx val val Key col col xxx val Val xxx val Val Key col col xxx val val xxx val val
Region 1 Region 2 Region 3 Region 4 ClientRegion assignment, Data Definition
Language operation (create, delete) are handled by HMaster Assigning and re-assigning
regions for recovery or load balancing and monitoring all servers Region Server 1 Region
Server 2 HMaster create, delete, update table
56. [56.](#) HBase Architectural Components - HMaster Key col col xxx val val xxx val val Key col col
xxx val val xxx val val Key col col xxx val Val xxx val Val Key col col xxx val val xxx val val
Region 1 Region 2 Region 3 Region 4 ClientRegion assignment, Data Definition
Language operation (create, delete) are handled by HMaster Assigning and re-assigning
regions for recovery or load balancing and monitoring all servers Region Server 1 Region
Server 2 HMaster create, delete, update table Monitors region servers

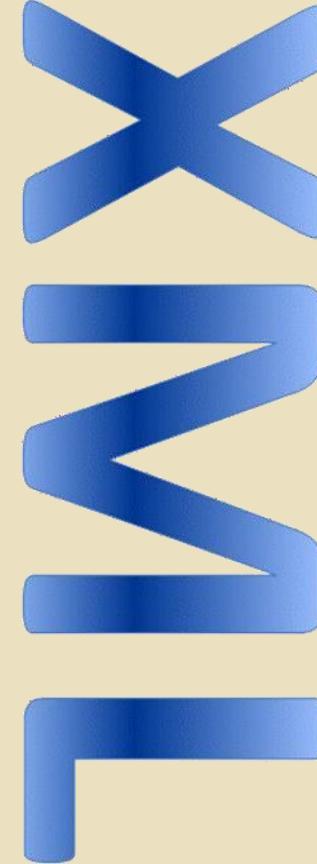
57. [57.](#) HBase Architectural Components - HMaster Key col col xxx val val xxx val val Key col col xxx val val xxx val val Key col col xxx val Val xxx val Val Key col col xxx val val xxx val val Region 1 Region 2 Region 3 Region 4 ClientRegion assignment, Data Definition Language operation (create, delete) are handled by HMaster Assigning and re-assigning regions for recovery or load balancing and monitoring all servers Region Server 1 Region Server 2 HMaster create, delete, update table Monitors region servers Assigns regions to region servers HBase has a distributed environment where HMaster alone is not sufficient to manage everything. Hence, ZooKeeper was introduced Assigns regions to region servers
58. [58.](#) Inactive HMaster HBase Architectural Components - ZooKeeper Key col col xxx val val xxx val val Key col col xxx val val xxx val val Key col col xxx val Val xxx val Val Key col col xxx val val xxx val val Region 1 Region 2 Region 3 Region 4 ZooKeeper is a distributed coordination service to maintain server state in the cluster Zookeeper maintains which servers are alive and available, and provides server failure notification Region Server 1 Region Server 2 Active HMaster ZooKeeper Active HMaster sends a heartbeat signal to ZooKeeper indicating that its active
59. [59.](#) Inactive HMaster HBase Architectural Components - ZooKeeper Key col col xxx val val xxx val val Key col col xxx val val xxx val val Key col col xxx val Val xxx val Val Key col col xxx val val xxx val val Region 1 Region 2 Region 3 Region 4 ZooKeeper is a distributed coordination service to maintain server state in the cluster Zookeeper maintains which servers are alive and available, and provides server failure notification Region Server 1 Region Server 2 Active HMaster heartbeat Region servers send their status to ZooKeeper indicating they are ready for read and write operation ZooKeeper
60. [60.](#) Inactive HMaster HBase Architectural Components - ZooKeeper Key col col xxx val val xxx val val Key col col xxx val val xxx val val Key col col xxx val Val xxx val Val Key col col xxx val val xxx val val Region 1 Region 2 Region 3 Region 4 ZooKeeper is a distributed coordination service to maintain server state in the cluster Zookeeper maintains which servers are alive and available, and provides server failure notification Region Server 1 Region Server 2 Active HMaster heartbeat Inactive server acts as a backup. If the active HMaster fails, it will come to rescue ZooKeeper
61. [61.](#) How the components work together? Key col col xxx val val xxx val val Key col col xxx val val xxx val val Key col col xxx val Val xxx val Val Key col col xxx val val xxx val val Region 1 Region 2 Region 3 Region 4 Region Server 1 Region Server 2 HMaster ZooKeeper 1

master is active • Active HMaster selection • Region Server session Active HMaster and Region Servers connect with a session to ZooKeeper

62. [62.](#) How the components work together? Key col col xxx val val xxx val val Key col col xxx val val xxx val val Key col col xxx val Val xxx val Val Key col col xxx val val xxx val val Region 1 Region 2 Region 3 Region 4 Region Server 1 Region Server 2 HMaster heartbeat 1 master is active • Active HMaster selection • Region Server session Active HMaster and Region Servers connect with a session to ZooKeeper ZooKeeper
63. [63.](#) How the components work together? Key col col xxx val val xxx val val Key col col xxx val val xxx val val Key col col xxx val Val xxx val Val Key col col xxx val val xxx val val Region 1 Region 2 Region 3 Region 4 Region Server 1 Region Server 2 HMaster heartbeat 1 master is active Ephemeral node Ephemeral node • Active HMaster selection • Region Server session ZooKeeper maintains ephemeral nodes for active sessions via heartbeats to indicate that region servers are up and running ZooKeeper
64. [64.](#) Introduction to HBaseHBase Read or Write
65. [65.](#) HBase Read or Write ZooKeeper .META location is stored in ZooKeeper There is a special HBase Catalog table called the META table, which holds the location of the regions in the cluster Here is what happens the first time a client reads or writes data to HBase Client Region Server Region Server DataNode DataNode The client gets the Region Server that hosts the META table from ZooKeeper Request for Region Server
66. [66.](#) HBase Read or Write ZooKeeper .META location is stored in ZooKeeper There is a special HBase Catalog table called the META table, which holds the location of the regions in the cluster Here is what happens the first time a client reads or writes data to HBase Client Region Server Region Server DataNode DataNode Meta table location The client gets the Region Server that hosts the META table from ZooKeeper Request for Region Server
67. [67.](#) HBase Read or Write ZooKeeper There is a special HBase Catalog table called the META table, which holds the location of the regions in the cluster Here is what happens the first time a client reads or writes data to HBase Client Meta Cache Region Server Region Server DataNode DataNode The client will query the .META server to get the region server corresponding to the row key it wants to access The client caches this information along with the META table location Meta table location Request for Region Server Get region server for row key from meta table .META location is stored in ZooKeeper

68. [68.](#) HBase Read or Write ZooKeeper There is a special HBase Catalog table called the META table, which holds the location of the regions in the cluster Here is what happens the first time a client reads or writes data to HBase Client Region Server Region Server DataNode DataNode Put row Meta Cache It will get the Row from the corresponding Region Server Get region server for row key from meta table .META location is stored in ZooKeeper Meta table location Request for Region Server Get row
69. [69.](#) Introduction to HBaseHBase Meta Table
70. [70.](#) HBase Meta Table Meta Table Row key value table, key, region region server Key col col xxx val val xxx val val Key col col xxx val val xxx val val Region 1 Region 2 Region Server Key col col xxx val val xxx val val Key col col xxx val val xxx val val Region 3 Region 4 Region Server Special HBase catalog table that maintains a list of all the Region Servers in the HBase storage system META table is used to find the Region for a given Table key
71. [71.](#) Introduction to HBaseHBase Write Mechanism
72. [72.](#) HBase Write Mechanism WAL Region Server Region MemStore MemStore HFile HFile HDFS DataNodeClient 1 When client issues a put request, it will write the data to the write-ahead log (WAL)1 Write Ahead Log (WAL) is a file used to store new data that is yet to be put on permanent storage. It is used for recovery in the case of failure.
73. [73.](#) HBase Write Mechanism WAL Region Server Region MemStore MemStore HFile HFile HDFS DataNodeClient 1 2 Once data is written to the WAL, it is then copied to the MemStore2 MemStore is the write cache that stores new data that has not yet been written to disk. There is one MemStore per column family per region.
74. [74.](#) HBase Write Mechanism WAL Region Server Region MemStore MemStore HFile HFile HDFS DataNodeClient 1 3 ACK 2 Once the data is placed in MemStore, the client then receives the acknowledgment3
75. [75.](#) HBase Write Mechanism WAL Region Server Region MemStore MemStore HFile HFile HDFS DataNodeClient 1 3 ACK 2 4 4 When the MemStore reaches the threshold, it dumps or commits the data into a HFile4 Hfiles store the rows of data as sorted KeyValue on disk
76. [76.](#) Introduction to HBaseDemo on HBase
77. [77.](#) Key Takeaways

eXtensible Markup Language



Software and hardware
independent database

Based on the lecture of Vörös Péter

http://people.inf.elte.hu/vopraai/korszeru_xml/

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body lang="en">Don't forget me this
        weekend!</body>
</note>
```

What is XML?

Data description language

Hardware and Software independent

Has nothing to do with the visualization
(In contrast with HTML, where this is the aim)

Building elements

- Elements
- Attributes
- Entities
- Namespaces
- XML declaration
- Processing Instruction
- Comments
- CDATA sections

Element in an XML document is a part between an opening and a corresponding closing tag

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body lang="en">Don't forget me this
weekend!</body>
</note>
```

In XML further informations can be described by attributes about an element.

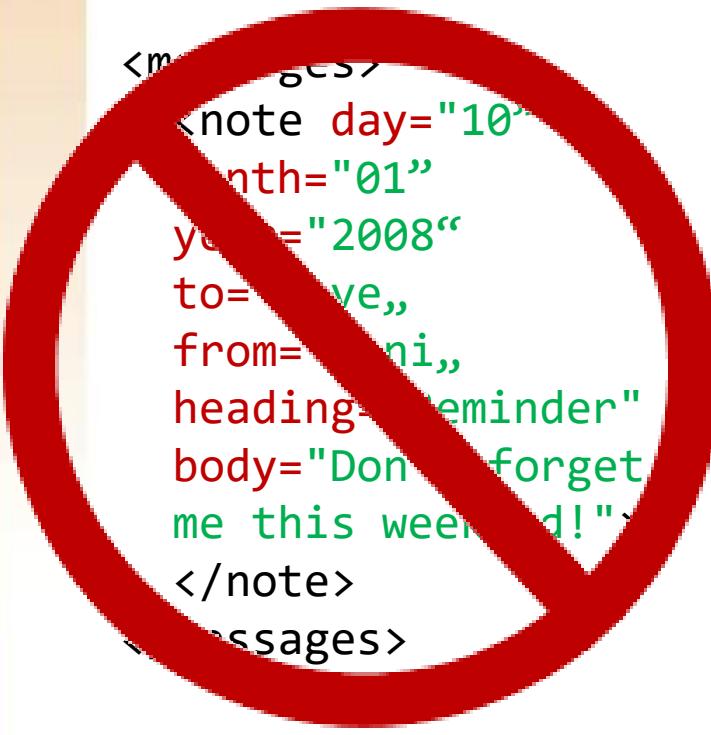
Attribute can only appear in an opening tag and the value must be quoted

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body lang="en">Don't forget me this
weekend!</body>
</note>
```

Too much attributes

```
<messages>
  <note day="10"
month="01"
year="2008"
to="Tove,,
from="Jani,,"
heading="Reminder"
body="Don't forget
me this weekend!">
  </note>
</messages>
```

Instead of using attributes, using elements gives a better overview



```
<messages>
  <note day="10"
        month="01"
        year="2008"
        to="Tove,"
        from="Jani,"
        heading="Reminder"
        body="Don't forget
               me this weekend!">
  </note>
</messages>
```

```
<note>
  <date>
    <year>2008</year>
    <month>01</month>
    <day>10</day>
  </date>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this
weekend!</body>
</note>
```

Do not use too much attributes, leave them for metadata

```
<messages>
  <note id="501">
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
  </note>
  <note id="502">
    <to>Jani</to>
    <from>Tove</from>
    <heading>Re: Reminder</heading>
    <body>I will not</body>
  </note>
</messages>
```

Entity

Entity is a shortening, to name a complicated or long textual data.

Entities can be defined freely in the document, can be used anywhere

When analysing, the entity is replaced by the text, that it shortens.

<	<
>	>
&	&
'	'
"	"

Name collision

Document 1

```
<table>
  <tr>
    <td>Apple</td>
    <td>Pear</td>
  </tr>
</table>
```

Document 2

```
<table>
  <name>Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```



N
A
M
E

C
O
L
L
I
S
I
O
N

Namespaces and qualified names

Namespaces are nominating names to make tags unique so that name collisions can be avoided

Nominating names made unique by namespaces are qualified names



Name collision

Document 1

```
<table>
  ...
</table>
```

Document 2

```
<table>
  ...
</table>
```

```
<root>
<h:table xmlns:h="http://www.w3.org/
TR/html4/">
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table xmlns:f="http://www.w3schools.com/furniture">
  <f:name>African Coffee
Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
</root>
```

XML Declaration

Optional in case of version 1.0,
mandatory in case of 1.1

If not given, then it is identified as 1.0 by
default

```
<?xml version="1.0" encoding="UTF-8"?>
```

We can add visualization information
to the document

```
<?xml-stylesheet type="text/css"  
 href="cd_catalog.css"?>
```

Processing instructions

```
<!-- This is just a comment  
-->
```

Visualization information is used rarely, not recommended by W3C, instead let's use XSLT.

Well formatted XML

1. Every tag must have a closing pair
(tag without text can be shortened)

```
<p>This is a paragraph.</p>  
<br />
```

2. Tags are case sensitive

```
<Message>incorrect</message>  
<message>correct</message>
```

3. Correct embedding

```
<b><i>This is incorrect</b></i>  
<b><i>This is correct</i></b>
```

Well formatted XML continued...

4. Every document must have exactly one root element, which contains all other elements

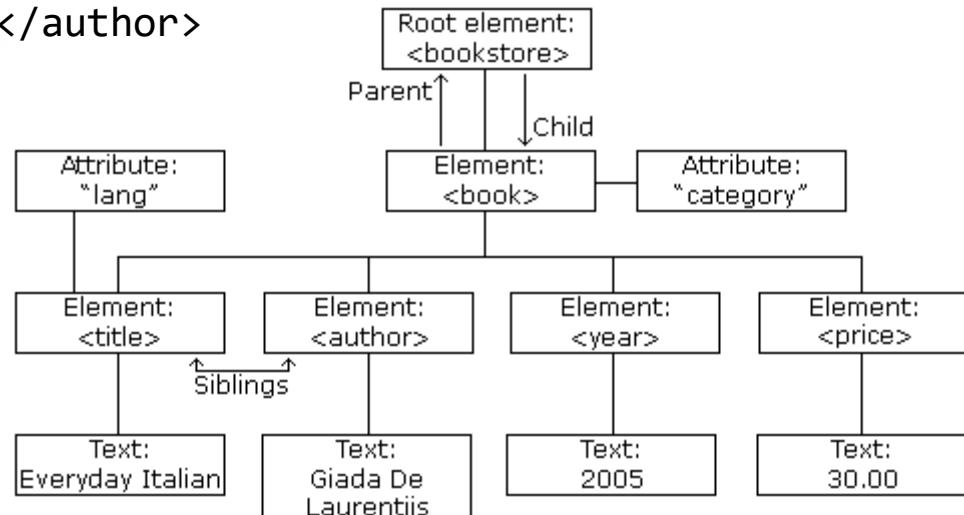
```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

5. We give attribute values in quotes

```
<title lang="English">HP</title>
<title lang="en">HP</title>
```

XML tree

```
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
</bookstore>
```



DTD (Document Type Description)

- A DTD schema descriptors define the structure of XML documents.
- With using DTD we specify, when an XML document is valid.
- DTD tells, when an element is legal and what embeddings are allowed.
- The expression power of the DTD is not enough to define types.

What DTD is good for?

- We can create special language with this (<http://www.w3.org/TR/html401/sgml/dtd.html>)
- Our application can easily check if a given data is valid.

How?

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

```
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

```
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

!DOCTYPE note gives that the root of the document
is note

!ELEMENT note gives that the note is build up by
„to,from,body” parts.

!ELEMENT to, from, heading, body give that those
elements are textual values

a	The element
e1?	0 or 1 occurrence of expression e1
e1*	0 or more occurrence of expression e1
e1+	1 or more occurrence of expression e1
e1, e2	e2 comes after e1
e1 e2	e1 or e2
(e)	Parentheses
#PCDATA	Text not to analized
EMPTY	No content
ANY	Any content
(#PCDATA a ₁ .. a _n)*	Mixed content

Limitations can be formulated for attributes too

#REQUIRED – Given attribute is required

```
<!ATTLIST person number ID #REQUIRED>
<person number="5" />
<person />
```

#IMPLIED – Given attribute is optional

```
<!ATTLIST contact fax CDATA #IMPLIED>
<contact fax="555-667788" />
<contact />
```

#FIXED – Value must be fix

```
<!ATTLIST sender company IDREF #FIXED "Microsoft">
<sender company="Microsoft" />
<sender company="W3Schools" />
```

Creating entity

```
<!ENTITY NEWSPAPER "Vervet Logic Times">  
<!ENTITY PUBLISHER "Vervet Logic Press">  
  
<body>&NEWSPAPER;&PUBLISHER;</body>
```

Validation based on DTD

An XML document is valid for a given DTD, if

- The document fits on the regular expressions
- Types of the attributes are correct
- Used identifier and references are correct

XSD - XML schema

Resembles to DTD

There are types

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

XSD example

```
<?xml version="1.0"?>
<xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xselement name="note">
  <xsccomplexType>
    <xssequence>
      <xselement name="date" type="xs:date"/>
      <xselement name="to,, minOccurs=,,1" maxOccurs="unbounded" .../>
      <xselement name="from" type="xs:string"/>
      <xselement name="heading" type="xs:string"/>
      <xselement name="body" type="xs:string"/>
    </xssequence>
    <xseattribute name="ID" type="xs:integer" use="required"/>
  </xsccomplexType>
</xselement>
</xsschema>
```

XPath

Expressions of XPATH must
be evaluated based on
the XML tree

- Navigate
- Select
- General information retrieval

Node

Node types of an XML tree
can be the following:

- Document
- Element
- Attribute
- Text
- Instruction
- Comment
- Namespace

Node

A node can has name, value
or both

- An element node has name,
but no value
- A text node has no name,
but has string value
- An Attribute node has
name and value

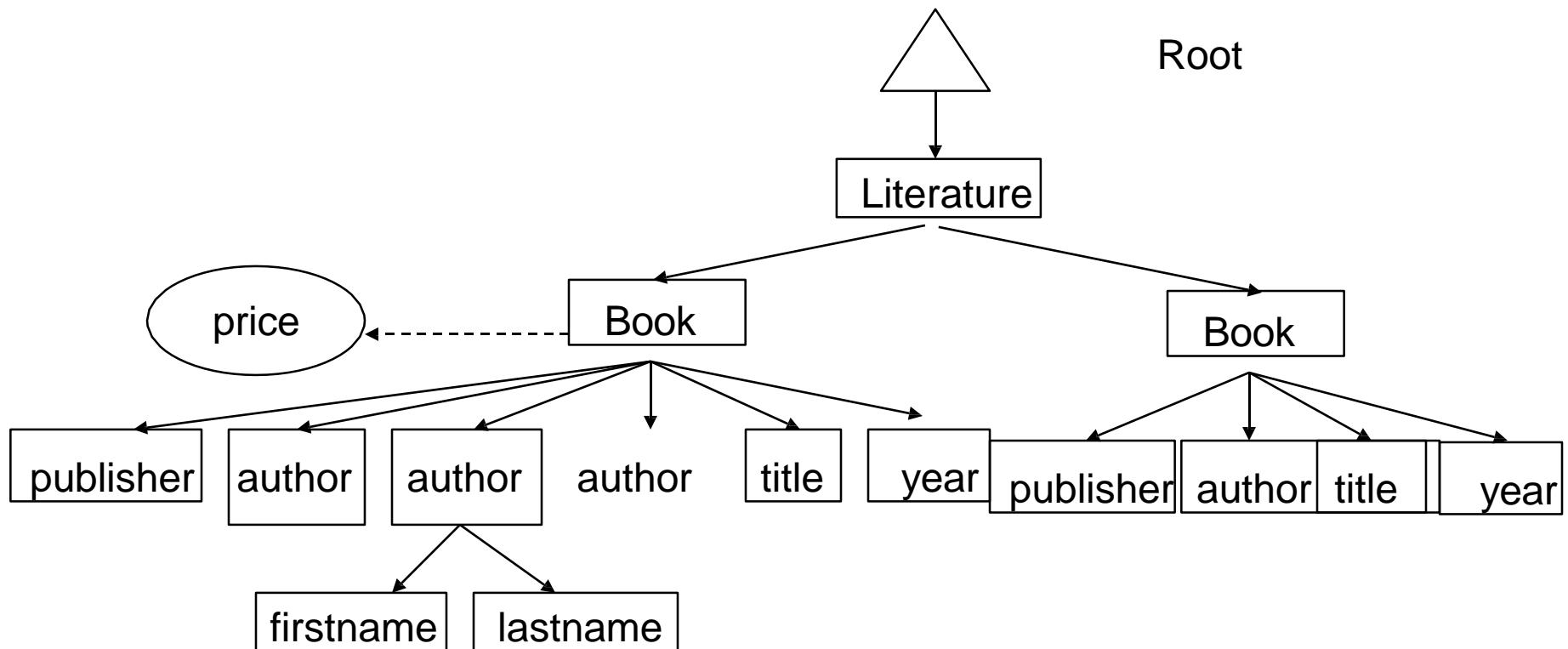
Node order

- Order of nodes is defined by the document order, which corresponds to pre-order admittance
 - A parent node precedes its children and attributes
 - Among the sibling nodes, attributes come first, then other types
 - Order of the attributes depends on implementation

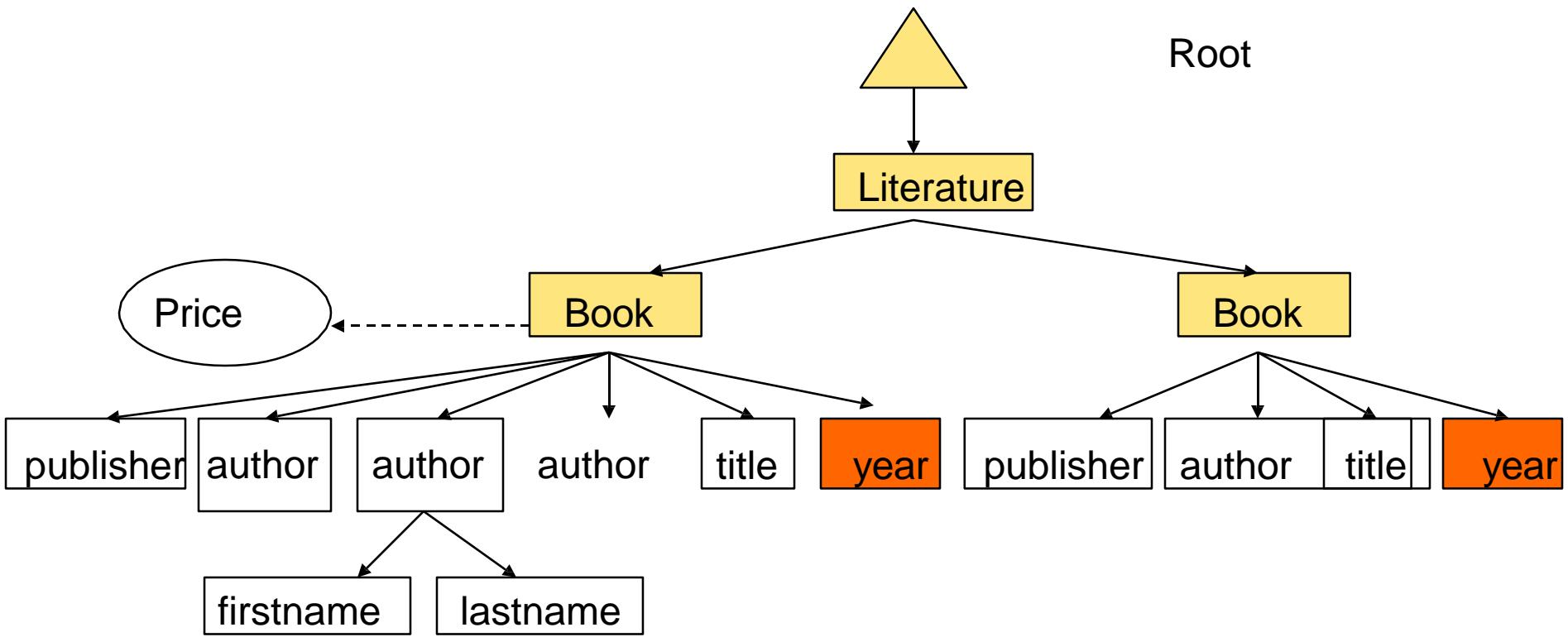
Example Xpath query

```
<literature>
  <book price='55'>
    <publisher>Addison-Wesley</publisher>
    <author>Serge Abiteboul</author>
    <author><firstname>Rick</firstname>
      <lastname>Hull</lastname>
    </author>
    <author>Victor Vianu</author>
    <title>Foundations of Databases</title>
    <year>1995</year>
  </book>
  <book> <publisher>Freeman</publisher>
    <author>Jeffrey D. Ullman</author>
    <title>Principles of Database and Knowledge Base
    Systems</title>
    <year>1998</year>
  </book>
</literature>
```

The XML tree



/literature/book/year



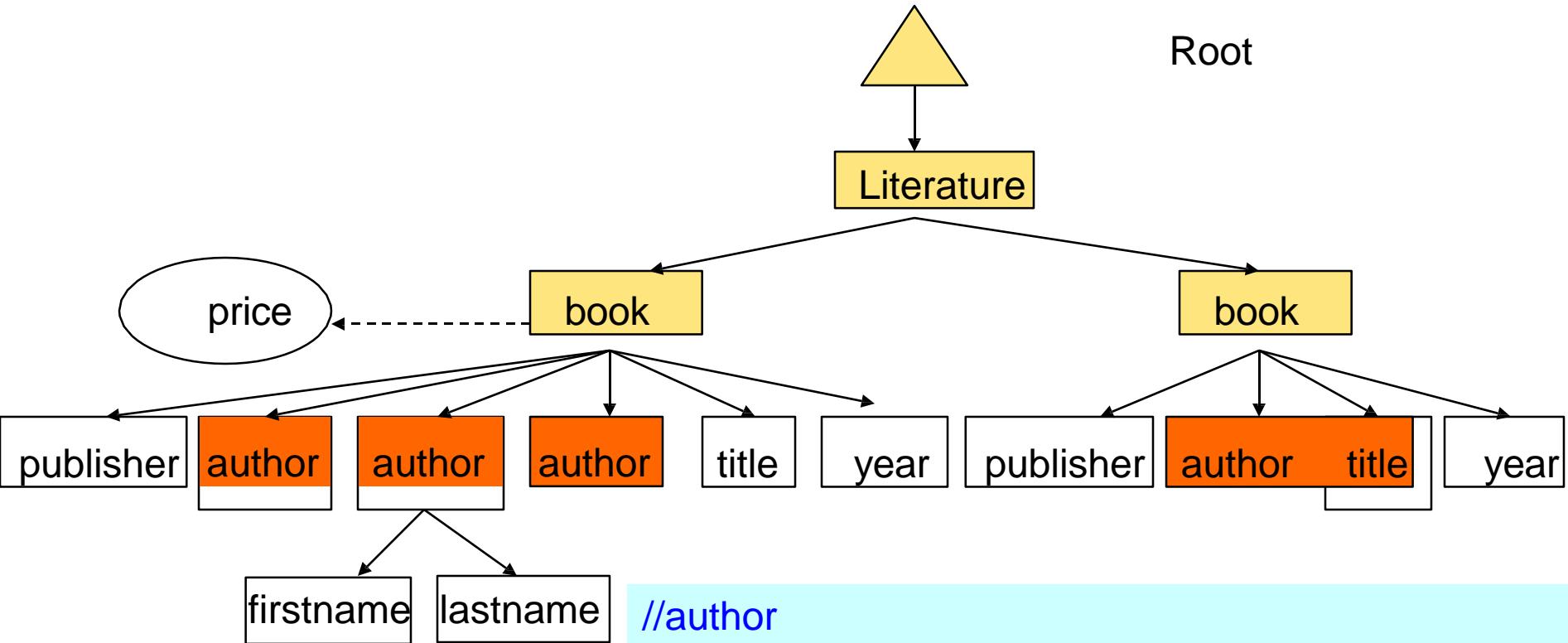
/literature/book/year

Result:

```
<year>1995</year>
<year>1998</year>
```

/literature/paper/year result is empty, because there is no paper.

//author

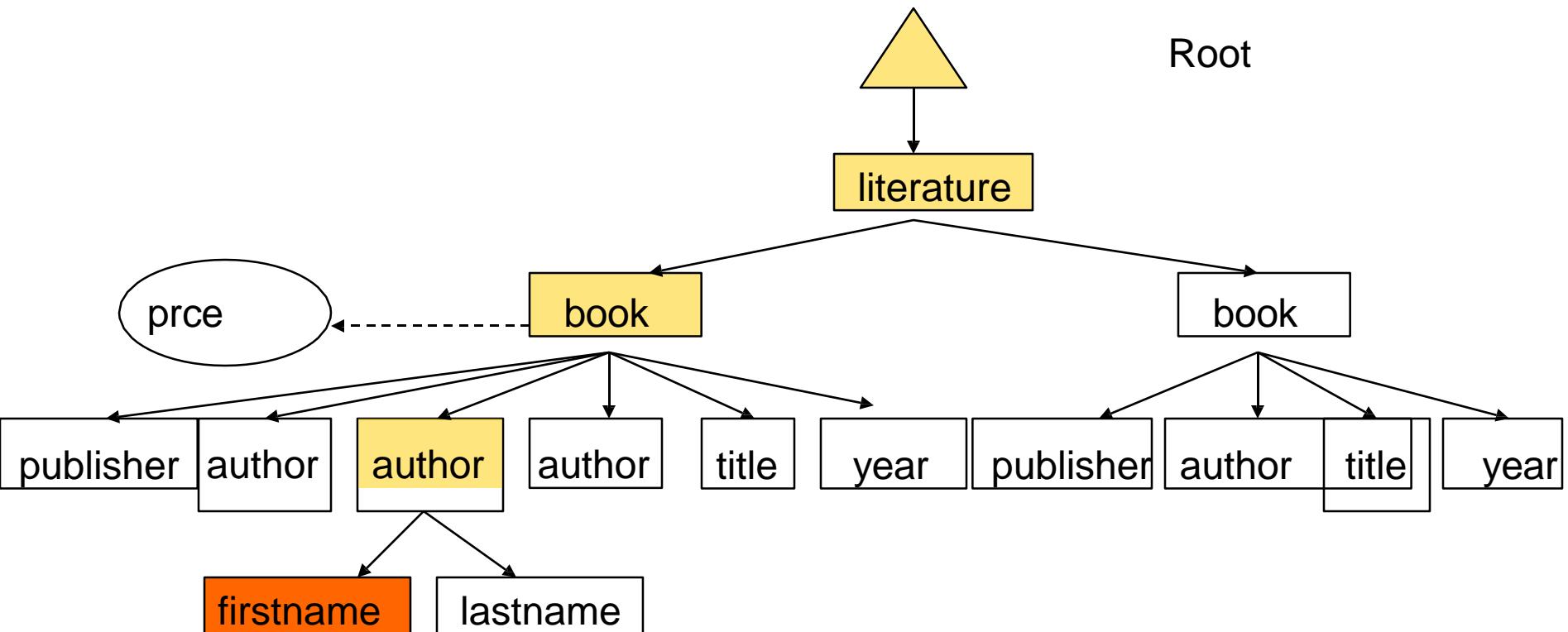


//author

Result:

```
<author>Serge Abiteboul</author>
<author><firstname>Rick</firstname>
      <lastname>Hull</lastname>
</author>
<author>Victor Vianu</author>
<author>Jeffrey D. Ullman</author>
```

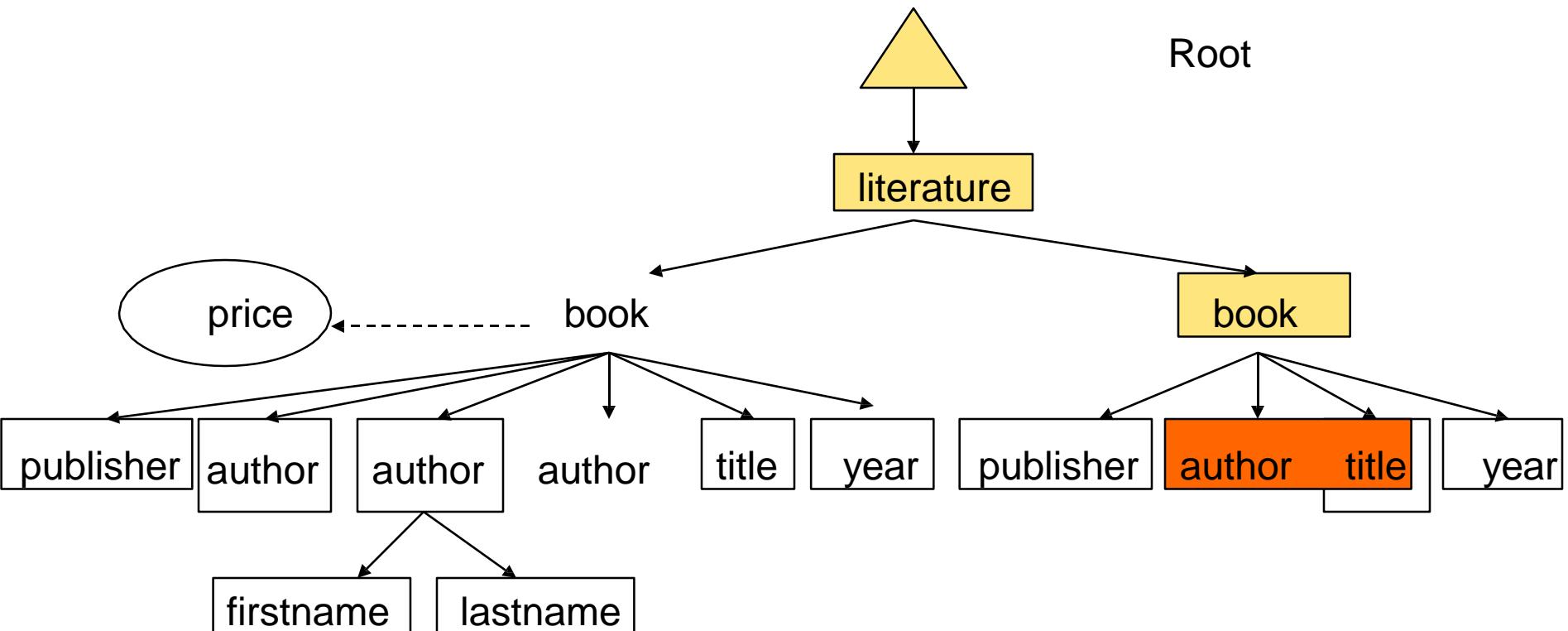
/literature//firstname



/literature//firstname

Result: <firstname>Rick</firstname>

/literature/book/author/text()



/literature/book/author/text()

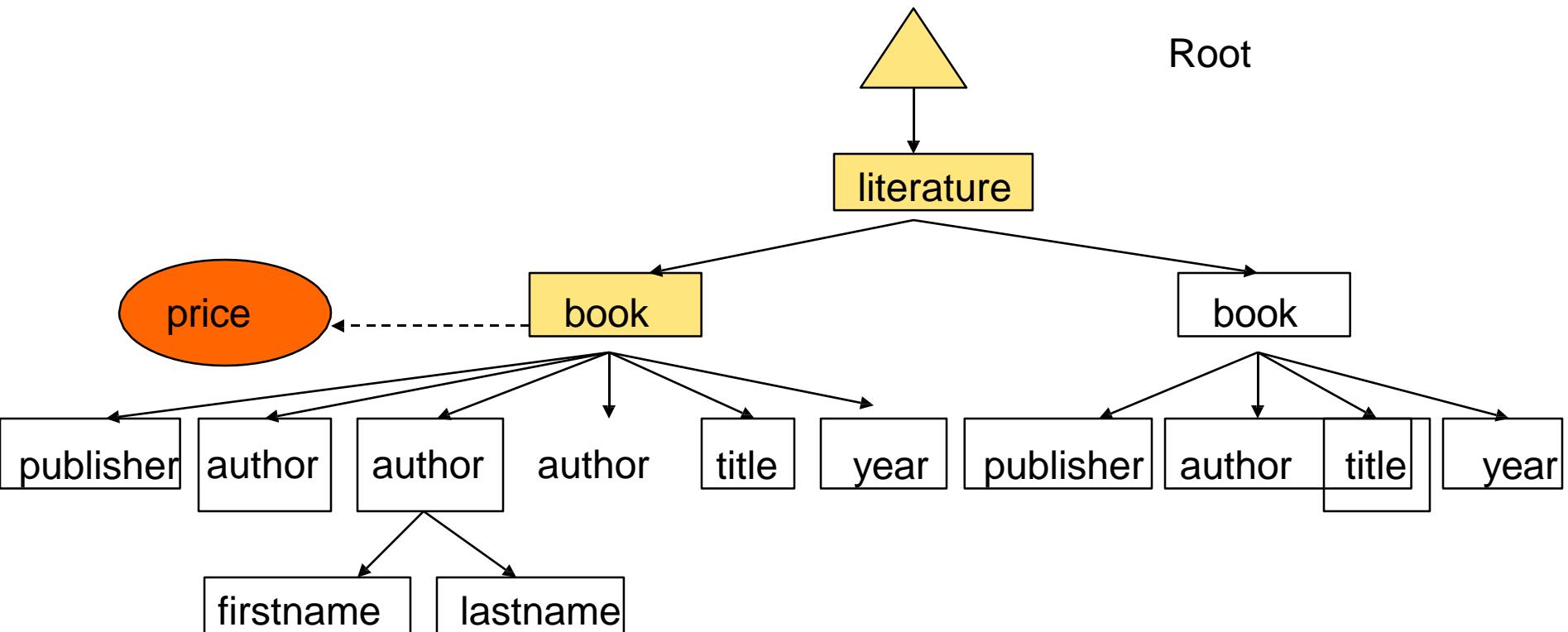
Result: Serge Abiteboul
 Victor Vianu
 Jeffrey D. Ullman

text() = gives back the value of text node

node() = gives back the node

name() = gives back the name of the nomination

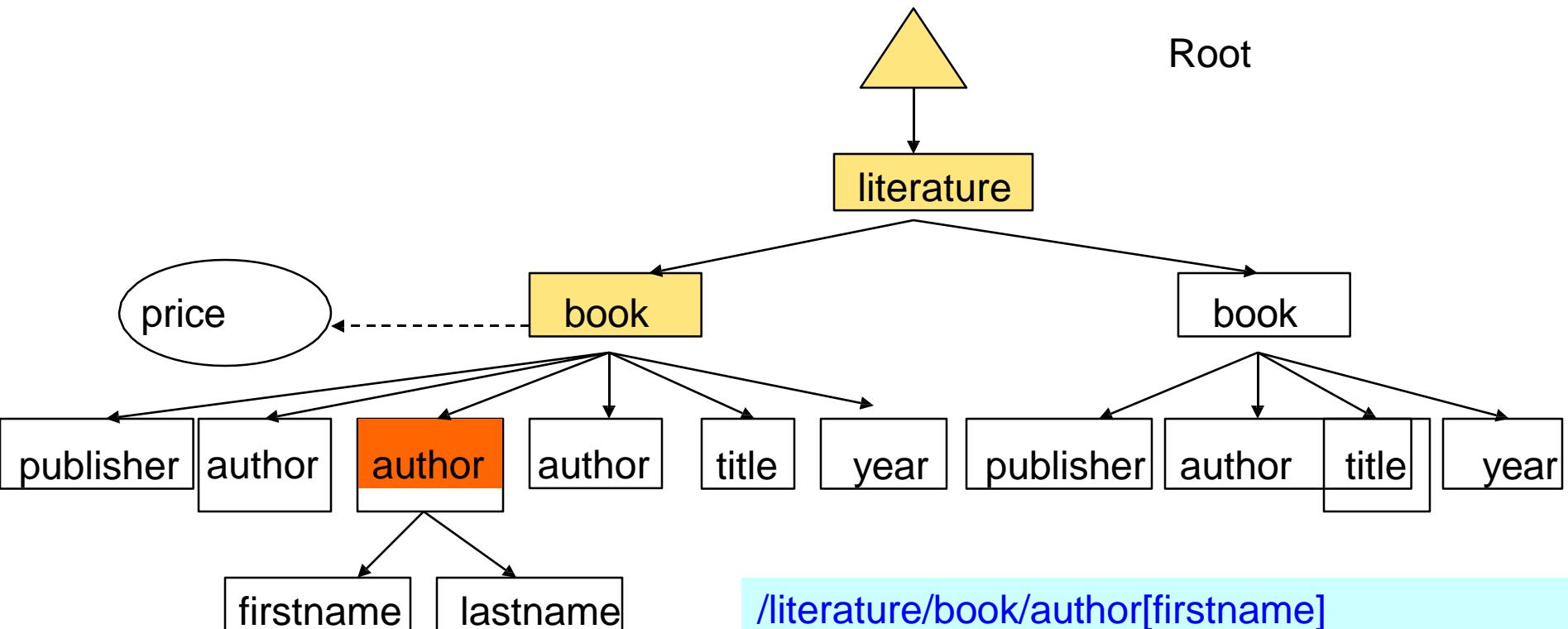
/literature/book/@price



/literature/book/@price

Result: 55

/literature/book/author[firstname]



/literature/book/author[firstname]

Result: <author>

<firstname>Rick</firstname>

<lastname>Hull</lastname>

</author>

[firstname] true, if the author element has
firstname child

Further examples

/literature/book[@price<60]

the price is smaller than 60

/literature/book[author/firstname= "Rick"]

firstname of one author is Rick

/literature/book[author/text()]

It has field given with text

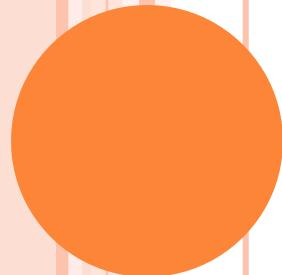
/literature/book[2]

the 2. book



Set operators

- Following set operators can be used:
 - | , union
 - intersect
 - except
- These clear multiple occurrences



XQUERY

Based on the
lecture of
Csubák Dániel

XQUERY - Introduction

- Declarative XML query language
 - XQuery is for XML what SQL for database tables
- Xquery has properties from programming languages and database languages too
 - Programming language properties:
 - Iteration, variables (*for...in*, *let...in*)
 - Recursive, user functions
 - Regular expressions, types
 - Lists, arrays
 - Database properties:
 - filtering
 - Grouping, joins



XQUERY - Introduction

- Uses Xpath expressions
- Possibilities to use:
 - Creating reports
 - Converting XML to XHTML
 - Creating queries



XQUERY - Example

- XML: books.xml
- Opening Document:
 - Calling doc("DOC_URI") function
- Using XPath expressions:
 - [doc\("http://www.w3schools.com/xpath/books.xml"\)/bookstore/book/title](http://www.w3schools.com/xpath/books.xml)
- Online XQuery evaluation
 - <http://www.xpathtester.com/xquery>



XQUERY - Example

- Using predicates
 - `doc("http://www.w3schools.com/xpath/books.xml")/bookstore/book[price<30]`

XQUERY - FLOWR

FLOWR expression built up by:

- iteration (**for**);
- defining variables (**let**);
- ordering result (**order**);
- using predicates (**where**);
- constructing result (**return**).



XQUERY - FLOWR

- `doc("books.xml")/bookstore/book[price>30]/title`
- Equivalent FLOWR:
 - `for $x in doc("books.xml")/bookstore/book
where $x/price>30
return $x/title`



XQUERY - FLOWR

- For
 - Usual
 - ```
for $x in (1 to 5)
return <test>{$x}</test>
```
  - Iterations can be counted with at keyword
    - ```
for $x at $i in doc("books.xml")/bookstore/book/title
return <book>{$i}. {data($x)}</book>
```
 - Can have multiple variables in the in part
 - ```
for $x in (10,20), $y in (100,200)
return <test>x={$x} and y={$y}</test>
```



# XQUERY - FLOWR

- Let

- With this we do not have to write the same thing multiple times
- Not giving back iteration (in contrast with for)
- ```
let $x := (1 to 5)
return <test>{$x}</test>
```



XQUERY - FLOWR

- Where

- We draw up condition regarding the result
- where \$x/price>30 and \$x/price<100



XQUERY - FLOWR

- Order by

- We describe the ordering of the result
- ```
for $x in doc("books.xml")/bookstore/book
 order by $x/@category, $x/title
 return $x/title
```



# XQUERY - FLOWR

- Return

- Specifies what we return
- ```
for $x in doc("books.xml")/bookstore/book
return $x/title
```



XQUERY – FLOWR + HTML

- Let's shape the output from XML to HTML

- ```

{
for $x in doc("books.xml")/bookstore/book/title
order by $x
return {$x}
}

```



# XQUERY – FLOWR + HTML

- Let's clear the tags

- ```
<ul>
{
for $x in doc("books.xml")/bookstore/book/title
order by $x
return <li>{data($x)}</li>
}
</ul>
```

XQUERY – join

- Using for on two different xml

```
<amazon>{
  for $a in doc("alexandra.xml")/book,
    $l in doc("libri.xml")/book
  where $a/@isbn = $l/@isbn
  return
    <book>
      { $a/title }
      <alexandra_price>{
        $a/price}</alexandra_price>,
      <libri_price>{ $l/price }</libri_price>
    </book>
}</amazon>
```



XQUERY – grouping

- There is no GROUP BY in XQuery language.

```
<result>{
  for $s in distinct-
    values(doc('literature.xml')/literature/book[publisher='P
      AND M']/author)
  return
    <author>{
      $s,
      for $c in
        document(, 'literature.xml')/literature/book[author=
          $s]/title
        return { $c }
    }</author>
}</result>
```



XQUERY – AGGREGATION

- How many books have been published by the authors per years? (Not showing nulls)

```
for $s in distinct-
  values(doc("http://www.libri.hu")/literature/book/author),
  $y in distinct-
  values(doc("http://www.libri.hu")/literature/book/@year)
let $k := doc("http://www.libri.hu")/literature/book[author=$s
  and @year=$y]
return
  if(exists($k)) then
    <result> {
      $s,
      <year> $y </year>,
      <count> count($k) </count>
    } </result>
  else ()
```

XQUERY – functions

- With the built in function of XQuery we can create complicate calculations. New functions can be defined too.

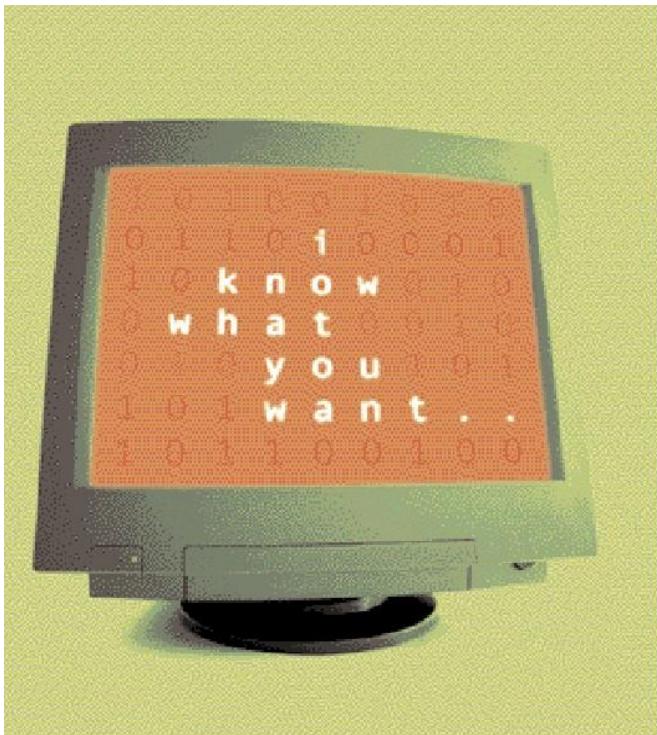
Function	Example
count	count((0,4,2)) → 3
max	max((0,4,2)) → 4
subsequence	subsequence((1,3,5,7),2,3) → (3,5,7)
empty	empty((0,4,2)) → false()
exists	exists((0,4,2)) → true()
distinct-values	distinct-values((4,4,2,4)) → (4,2)
to	(1 to 10)[. mod 2 eq 1] → (1,3,5,7,9)

REFERENCES

- XPath
 - <http://www.w3schools.com/xpath/default.asp>
- XQuery
 - <http://www.w3schools.com/xquery/>



Semantic Web



THE SEMANTIC WEB

A new form of Web content
that is meaningful to computers
will unleash a revolution of new abilities

by

TIM BERNERS-LEE,
JAMES HENDLER and
ORA LASSILA

Lehotay-Kéry Péter
based on lecture of
Gombos Gergő

Overview

- History of Semantic Web
- Language of Semantic Web
 - XML
 - RDF(S)
 - OWL
- SPARQL
- Ontologies
- Semantic Web Services
- Federated queries
- Semantic Web Applications

The Semantic Web

“The **Semantic Web** is an extension of the current web in which information is given well-defined **meaning**, better enabling computers and people to **work in co-operation**.“

[Berners-Lee *et al*, 2001]



Web nowadays

- Tipical usage:
 - Information sharing
 - Searching for people, product, etc
- Most web page is readable for users.

Searching limitations

- Searchings work from stored data.
- Results are very sensitive for the given words.
- Result is a web page.
- Reason:
 - Most content is not structured. Connections, logical deduction is hard between them.

What is Web of Data?

- 1994
- HTML, URI
- Formalizing language and connections between pages.
- File level connections

The screenshot shows the W3C homepage from September 2004. The main navigation bar includes links for Activities, Technical Reports, Site Index, New Visitors, About W3C, Join W3C, and Contact W3C. A banner at the top reads "Leading the Web to Its Full Potential...". The "Semantic Web Activity" section is highlighted with a blue arrow pointing to it. This section contains news items about the 26th Internationalization & Unicode Conference and Dead Works Adaptations. It also features a "Technology and Society domain" and a "Members" section. The "News" and "Search" sections are also visible on the right.

Semantic Web

The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. It is a collaborative effort led by W3C with participation from a large number of researchers and industrial partners. It is based on the Resource Description Framework (RDF), which integrates a variety of applications using XML for syntax and URIs for naming.

"The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation." -- Tim Berners-Lee, James Hendler, Ora Lassila, *The Semantic Web*, Scientific American, May 2001

On this page: Activity Statement | Specifications | Publications | Presentations | Groups

Nearby: Advanced Development | SWAD Europe | Semantic Web Coordination | RDF | RDF Core | RDF Data Access | RDF Ontology | Best Practices and Deployment | Interest Group | Developer Tools

News and Events

- [RDF Data Access Use Cases and Requirements](#) Updated 2004-08-04. The RDF Data Access Working Group has released an updated Working Draft of RDF Data Access Use Cases and Requirements. The draft suggests how an RDF query language and data access protocol could be used in the construction of novel, useful Semantic Web applications in areas like Web publishing, personal information management, transportation and tourism.
- [Representing Specified Values in OWL](#) 2004-08-03. The Semantic Web Best Practices and Deployment (SWBP) Working Group has released the First Public Working Draft of [Representing Specified Values in OWL](#), "value partitions" and "value sets". Comments are welcome. The draft presents methods for representing modified values and collections of values in the [OWL Web Ontology Language](#).
- [Call for Participation: Public Workshop on Semantic Web for Life Sciences](#) 2004-07-28. Position papers are due 8 September for the W3C Workshop on Semantic Web for Life Sciences to be held in Cambridge, MA, USA on 27-28 October. Attendees will discuss how Semantic Web technologies such as [RDF](#), [OWL](#) and the [Life Sciences Identifier](#) (LSID) help to manage modern life sciences research, enable disease understanding and accelerate the development of therapies.
- [RDF Data Access Working Group Meets to Select Initial Design](#) 2004-07-26. The [RDF Data Access Working Group](#) reviewed use cases for ebXML and XQuery integration and selected an initial design at its second face-to-face meeting in Carlsbad, California, hosted by [Network Inference](#).

Syntactic Web

http://www2002.org

WWW 2002
THE ELEVENTH INTERNATIONAL WORLD WIDE WEB CONFERENCE

Sheraton Waikiki Hotel
Honolulu, Hawaii, USA
7-11 May 2002

CONFERENCE LOGO
International World Wide Web Conference Committee

1 LOCATION. 5 DAYS. LEARN. INTERACT.

Registered participants coming from:
Australia - Canada - Chile - Denmark - France - Germany - Ghana - Hong Kong - India - Ireland - Japan - Malta - New Zealand - The Netherlands - Norway - Singapore - Switzerland - The United States - Vietnam - Zambia

REGISTER NOW

On 7-11 May 2002, Honolulu, Hawaii will provide the backdrop for The Eleventh International World Wide Web Conference. This prestigious series of the International World Wide Web Conference Committee (IWC) attracts participants from around the world, and it provides a public forum for the World Wide Web Consortium (W3C) through the annual W3C track.

The conference is being organized by the International World Wide Web Conference Committee (IWC), the University of Hawaii and the Pacific Telecommunications Council (PTC).

FEATURED SPEAKERS (CONFIRMED)

Tim Berners-Lee, inventor of the World Wide Web and Director of the W3C who now holds the 3Com Founders chair at the Laboratory for Computer Science at the Massachusetts Institute of Technology (MIT).
Richard A. DeMillo, vice president and chief technology officer for Hewlett-Packard Company.
Ian Foster, guru of 'Grid Computing' associate professor at the University of Chicago and MacArthur Prize Winner.

Tim Berners-Lee - Netscape

Contents See also

Short Bio Before you mail me Address Talk articles &c Speaking engagements Press Interviews Longer Bio Slides from some talks Design Issues: web architecture World Wide Web Consortium Frequently Asked Questions Weaving the Web

Tim Berners-Lee

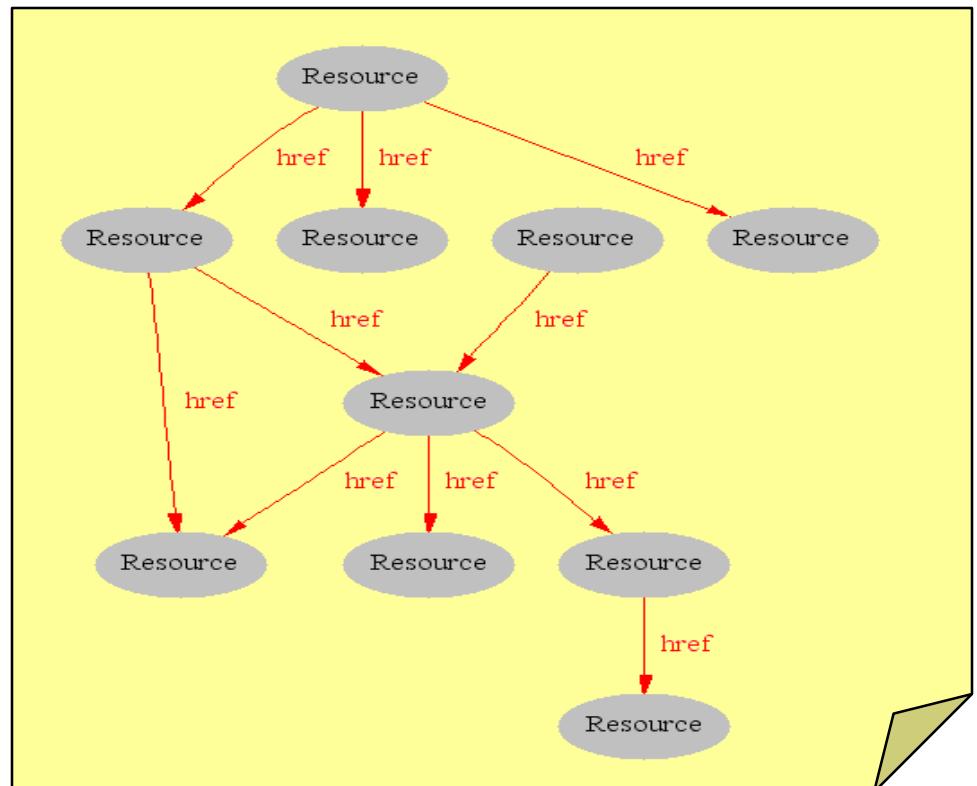
Weaving the Web by Tim Berners-Lee with Mark Weitzman, Harper San Francisco; Hardback; ISBN 0062515861, Abridged audio cassette abridged ISBN 0062512156 and various other languages.

Bio

A graduate of Oxford University, England, Tim now holds the 3Com Founders chair at the Laboratory for Computer Science and Artificial Intelligence (CSAIL) at the Massachusetts Institute of Technology (MIT). He directs the World Wide Web Consortium, an open forum of companies and organizations with the mission to lead the Web to its full potential.

With a background of system design in real-time communication and text processing software development, in 1989 he invented the World Wide Web, an Internet-based hypertext initiative for global information sharing, while working at CERN, the European Particle Physics Laboratory. He wrote the first web client (browser/editor) and server in 1990.

Before coming to CERN, Tim worked with Image Computer Systems, of Ferndown, Dorset, England and before that a



Syntactic Web

- A place, where
 - Computers make the visualization
 - People connect the web pages and interpret
- *Why not doing this computers?*

Why is it hard for computers?

- Let's look at a typical web page

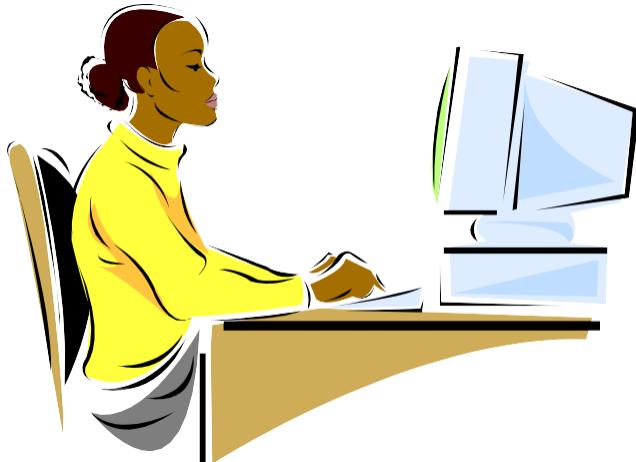
- Formatting:

- Visualizing styles (font size, colours)
 - Links to documents

- Knowledge can be (easily) interpreted for users, but not for computers

The screenshot shows the homepage of the WWW 2002 conference. At the top, there is a logo for 'WWW 2002 HAWAII' with the URL 'http://www2002.org'. The main title 'WWW 2002' is in large blue letters, followed by 'THE ELEVENTH INTERNATIONAL WORLD WIDE WEB CONFERENCE'. Below this, it says 'Sheraton Waikiki Hotel Honolulu, Hawaii, USA 7-11 May 2002'. To the right, there is a 'CONFERENCE ORGANIZERS' section with the International World Wide Web Conference Committee logo. A banner below the title reads '1 LOCATION. 5 DAYS. LEARN. INTERACT.' On the left, a sidebar lists navigation links: 'Conference Proceedings', 'Call for Participation Program', 'Registration Information', 'Hotel Accommodation', 'Conference Committee', 'Sponsorship/Exhibition Opportunities', 'Volunteer Information', 'Information about Hawaii', and 'Previous & Future WWW Conferences'. The main content area features a 'REGISTER NOW' button and information about the conference's global reach. At the bottom, there is a 'FEATURED SPEAKERS (CONFIRMED)' section with portraits and names of speakers like Tim Berners-Lee, Ian Foster, and Richard A. DeMillo.

Limits of web



The internet can be reached through Machine-to-Human approach and there are not many application that is able to apply Machine-to-Machine approach.

Aim

- Creating Web content, that can be interpreted on the level of computers.



XML

- Can be defined by user and domain specific

HTML:

```
<H1>Internet and World Wide Web</H1>
    <UL>
        <LI>Code: G52IWW
        <LI>Students: Undergraduate
    </UL>
```

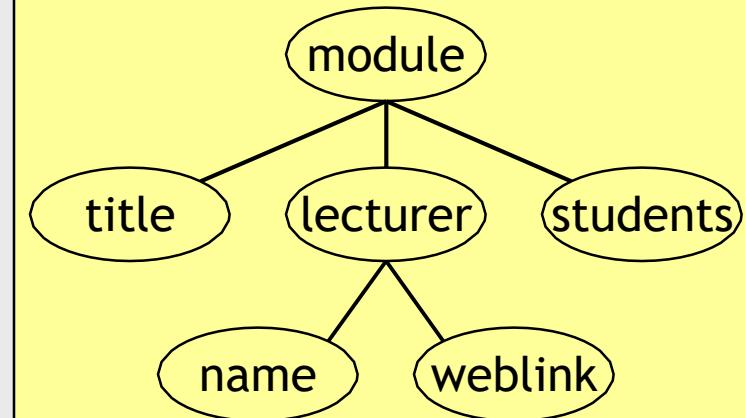
XML:

```
<module>
    <title>Internet and World Wide Web</title>
    <code>G52IWW</code>
    <students>Undergraduate</students>
</module>
```

XML: Document = labeled tree

- node = label + contents

```
<module date="...">
    <title>...</title>
    <lecturer>
        <name>...</name>
        <weblink>...</weblink>
    </lecturer>
    <students>...</students>
</module>
```



- DTD: grammatical and structural description for valid XML trees

XML

- Information in an XML-Document is clean by instinct
 - *For semantic notations*
 - Notations are domain-specific
- But machines do not have instinct
 - Tags are not contain any information for machines.
- DTD and XML Schema describes the structure of documents and not the knowledge inside the document.
- Flaw of XML is semantics

XML is only first step

- Semantic annotation
 - HTML \Rightarrow visualization
 - XML \Rightarrow content
- Metadata
 - Interpreted inside document
 - No bound on vocabulary
- **RDF** the next step

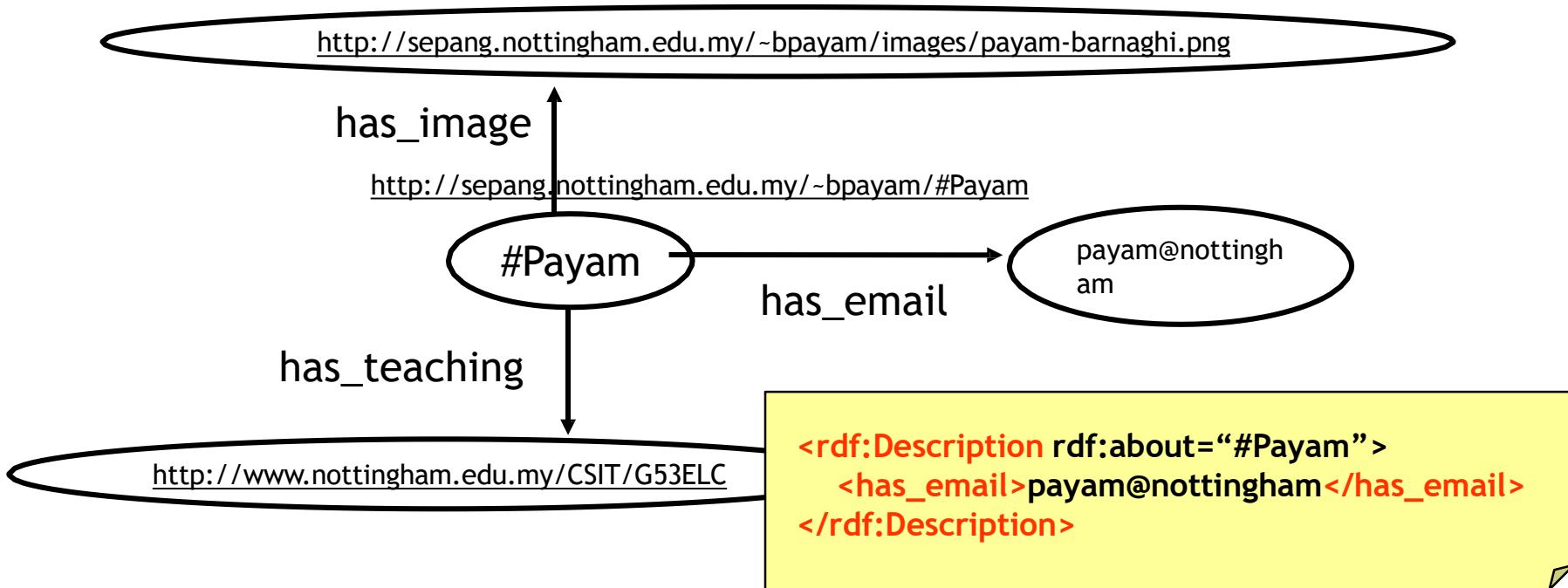
Resource Description Framework (RDF)

- W3C standard
- Connection between documents
- Contains triples:
 - <subject, predicate, object>
 - <“Mozart”, composed, “The Magic Flute” >
- RDFS extends RDF with basic “ontology vocabulary”:
 - Class, Property
 - Type, subClassOf
 - domain, range



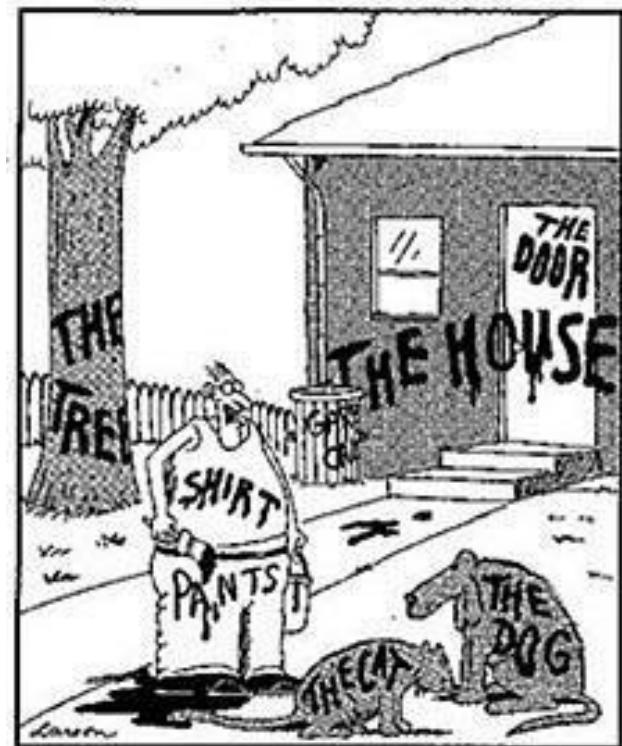
RDF semantic annotation

- Provides RDF metadata about Web content
- **Object -> Attribute-> Value triples**
- **XML syntax**
- Concatenated triplets form graphs



RDF: Base ideas

- Resources
 - Every resource represented by an URI (Universal Resource Identifier)
 - URI can be URL (web address) or other identifier
 - Every resource would be an object, that we want to describe
 - Books
 - Person
 - Places, etc.



"Now! ... That should clear up
a few things around here!"

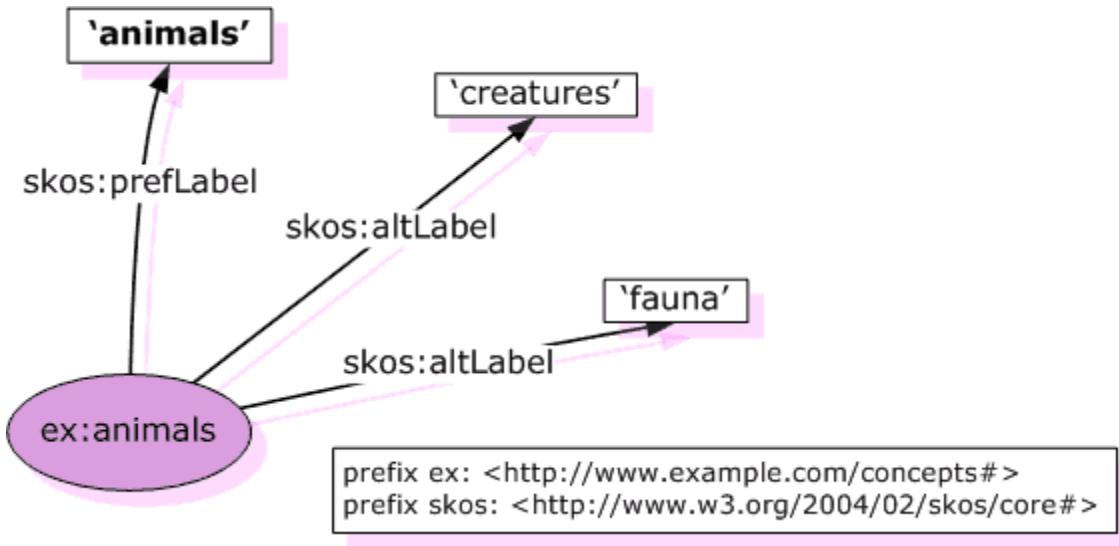
RDF: Base ideas

- Properties
 - Properties are special resources
 - Describing the connection between resources.
 - Eg: “written by”, “composed by”, “title”, “topic”, etc.
 - In RDF these are identified by URI.
- This will give a global naming schema.

RDF: Base ideas

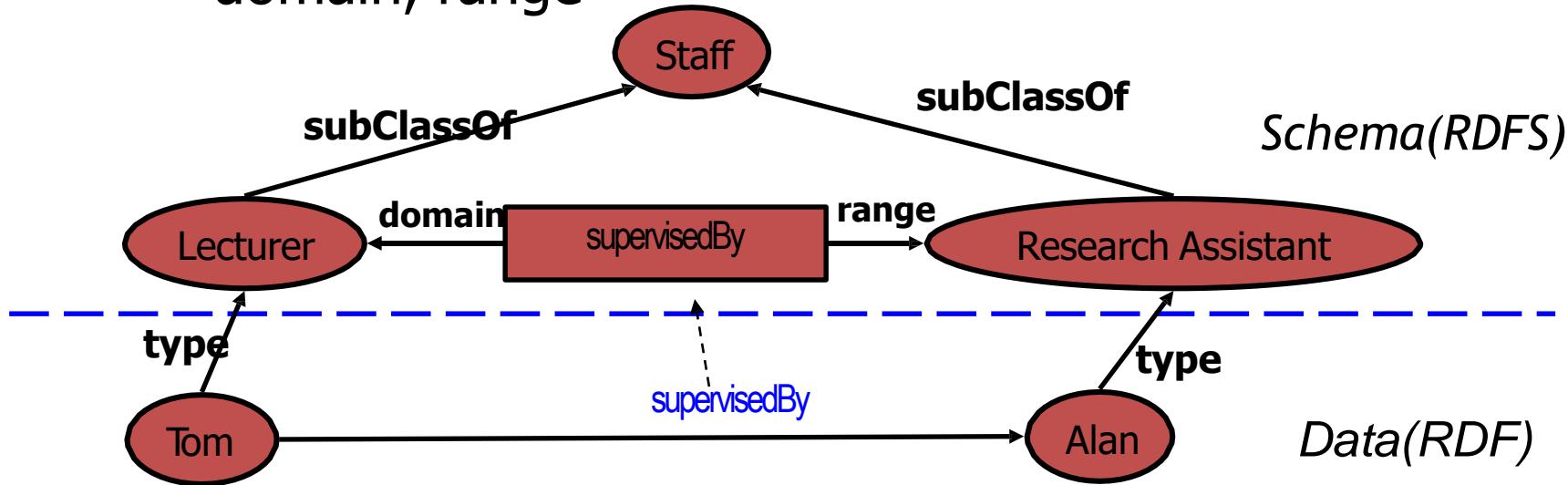
- Statements
 - Are subject-predicate-object triples
 - Object can contain resource, property or value.

RDF Example



What we get with RDF Schema?

- Vocabulary for RDF
- Vocabulary described in typed hierarchy
 - Class, subClassOf, type
 - Property, subPropertyOf
 - domain, range



RDF Formats

Feature	Expresses RDF 1.0 @prefix [] ; a	Collections	Numeric literals	Literal subj	RDF Path	Rules	Formulae
<i>syntax:</i>		(<a>)	2	7 a n:prime.	xly^z	{?x}=>{?x}	{ } @forAll
NTriples	y						
Turtle	y	y	y				
N3 RDF	y	y	y	y	y	y	
N3 Rules	y plus	y	y	y	y	y	
N3	y plus	y	y	y	y	y	y

RDF Formats

- XML

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"  
    xmlns:dc="http://purl.org/dc/elements/1.1/">  
    <rdf:Description rdf:about="http://en.wikipedia.org/wiki/Tony_Benn">  
        <dc:title>Tony Benn</dc:title>  
        <dc:publisher>Wikipedia</dc:publisher>  
    </rdf:Description>  
</rdf:RDF>
```

- N-Triples

```
<http://www.w3.org/TR/rdf-syntax-grammar> <http://purl.org/dc/elements/1.1/title>  
"RDF/XML Syntax Specification (Revised)" .  
<http://www.w3.org/TR/rdf-syntax-grammar> <http://example.org/stuff/1.0/editor> _:bnode .  
_:bnode <http://example.org/stuff/1.0/fullname> "Dave Beckett" .  
_:bnode <http://example.org/stuff/1.0/homePage> <http://purl.org/net/dajobe/> .
```

RDF Formats

- **Turtle**

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .  
@prefix dc: <http://purl.org/dc/elements/1.1/> .  
@prefix ex: <http://example.org/stuff/1.0/> .  
<http://www.w3.org/TR/rdf-syntax-grammar>  
    dc:title "RDF/XML Syntax Specification (Revised)" ;  
    ex:editor [  
        ex:fullname "Dave Beckett";  
        ex:homePage <http://purl.org/net/dajobe/>  
    ] .
```

- **N3(Notation3)**

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .  
<http://en.wikipedia.org/wiki/Tony_Benn>  
    dc:title "Tony Benn";  
    dc:publisher "Wikipedia".
```

Querying RDF data

- Query language: SPARQL.
 - Similar to SQL
- RDF is a directed, labeled graph data model to visualize information.
- Most queries contain triple samples
- In certain triples we can give variables.

SPARQL example

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
```

```
SELECT ?lat ?lon
WHERE {
    dbpedia:Loránd_Eötvös dbpedia-owl:birthPlace ?place .
    ?place geo:lat ?lat .
    ?place geo:long ?lon .
}
```

- More example:
<http://librdf.org/query>

SPARQL

- Modifiers:
 - Filter
 - Filtering the results
 - Optional
 - Optional information
 - Limit
 - Number of rows in result
 - Order by
 - Ordering result data
 - Distinct
 - Eliminating duplicates
 - Offset
 - Skip a specified number of solutions

SPARQL query forms

- SELECT: querying information
- ASK: is there a subgraph corresponding to the sample
- CONSTRUCT: building graph based on template
- DESCRIBE: describing a notion
- Data modification (SPARQL 1.1)
 - INSERT, UPDATE, DELETE

Ontologies

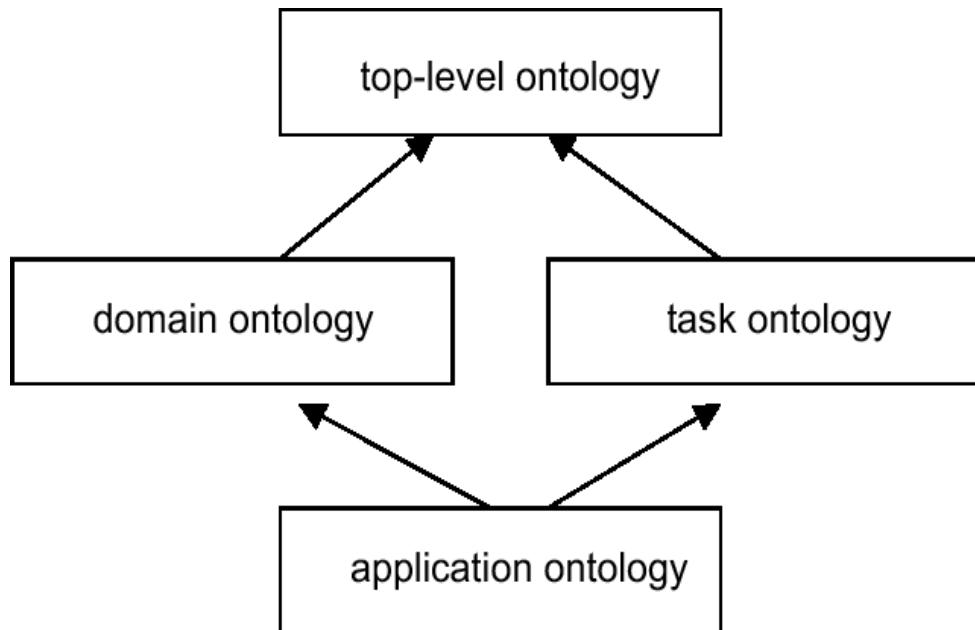
- Objects, notions, entities corresponding to an area
- vocabulary: compliance between notions of real world and URIs
- **ontology**: vocabulary + connections, limitations and rules defined between notions
- Example:
 - **Dublin Core** (dc): metadata of resources
 - **Friend of a Friend** (foaf): social networks
 - **RDF**: some built in notion

Types of Ontologies

[Guarino, 98]

Describe **very general concepts** like space, time, event, which are independent of a particular problem or domain. It seems reasonable to have unified top-level ontologies for large communities of users.

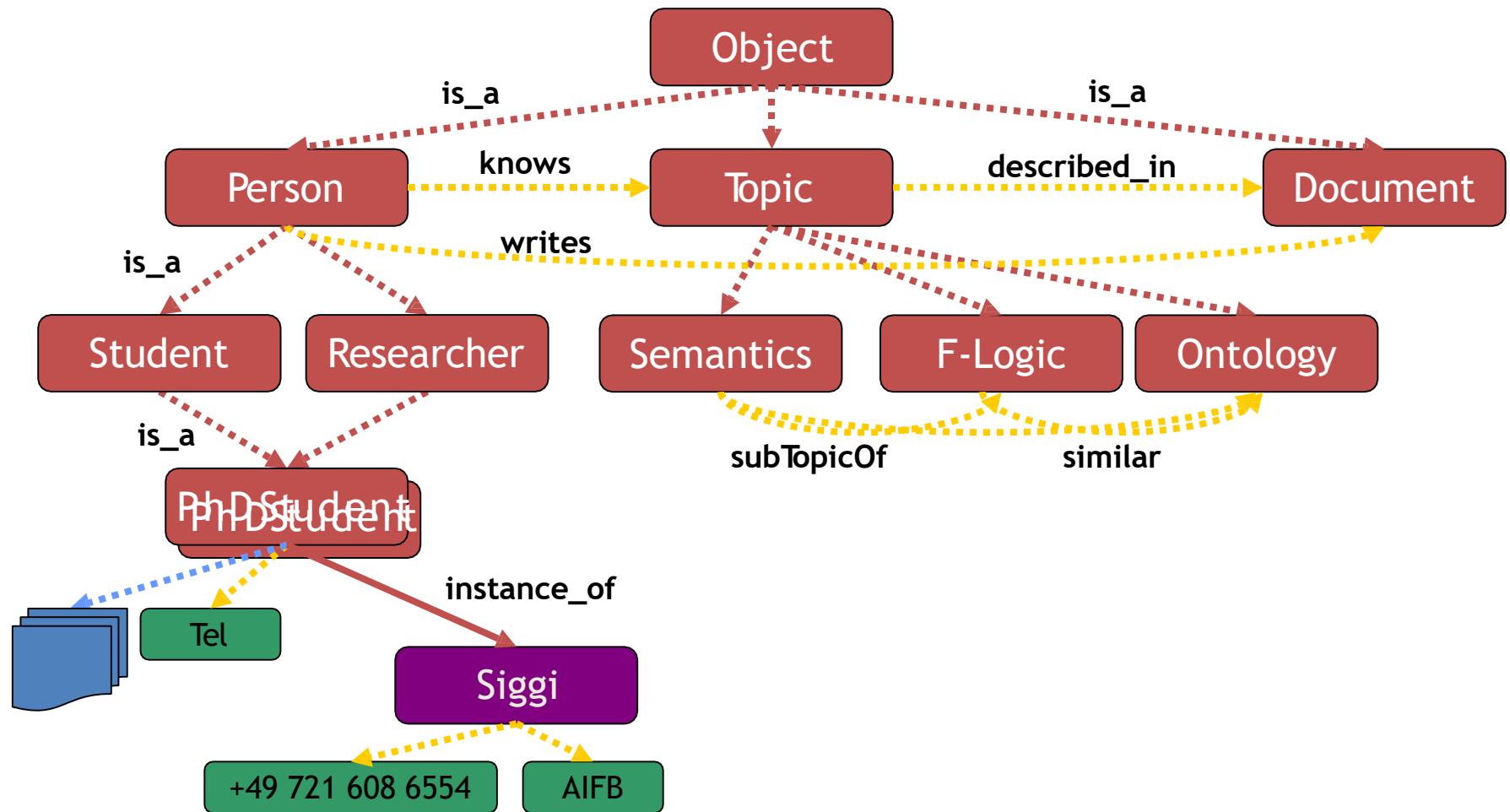
Describe the vocabulary related to a **generic domain** by specializing the concepts introduced in the top-level ontology.



Describe the vocabulary related to a **generic task or activity** by specializing the top-level ontologies.

These are the most specific ontologies. Concepts in application ontologies often correspond to **roles played by domain entities while performing a certain activity**.

A Sample Ontology



Ontologies (OWL)

- RDFS useful, but not enough
- Certain applications need information, that describe connections between classes.
- Example:
 - sameAs
 - symmetrical
 - disjointWith
 - etc
- This is how OWL (Web Ontology Language) born

RDFS vs OWL

- RDF Schema describes classes and properties with the hierarcic connection between them.
- OWL is a richer language which describes features between classes, properties.

Ontology and logic

- Based on given knowledge creating new.
- Examples:

X is author of Y \Rightarrow Y is written by X

X is parent of Y; Y is parent of Z \Rightarrow

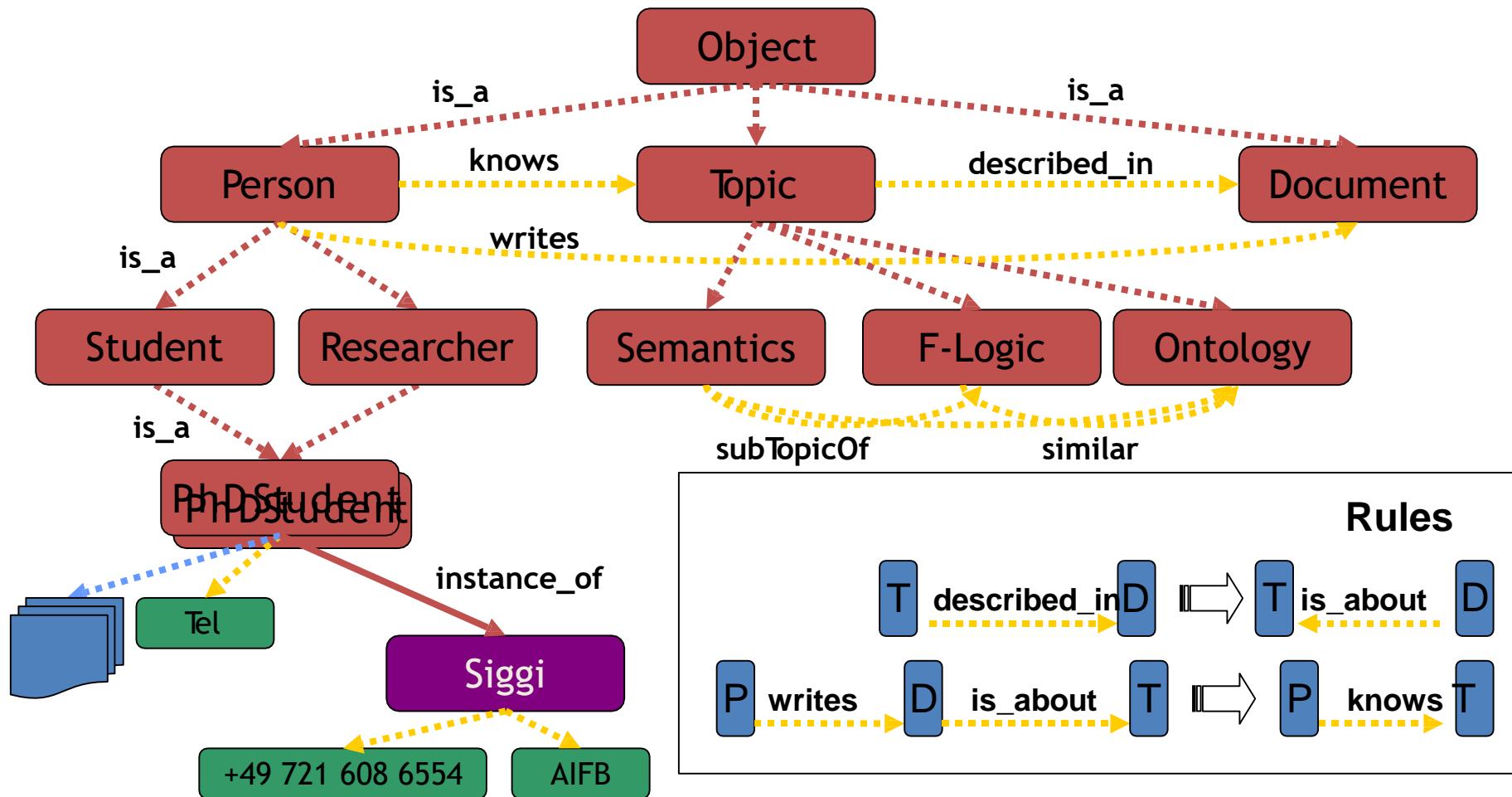
X is grandparent of Z

Cars are a kind of vehicle;

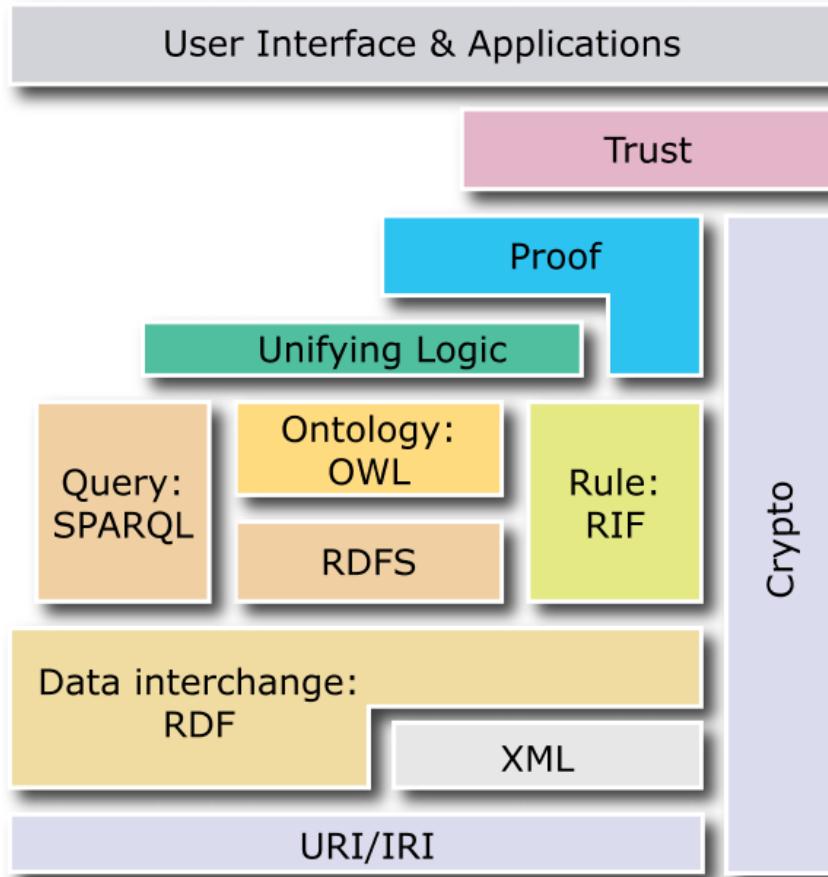
Vehicles have 2 or more wheels \Rightarrow

Cars have 2 or more wheels

Ontology and logic



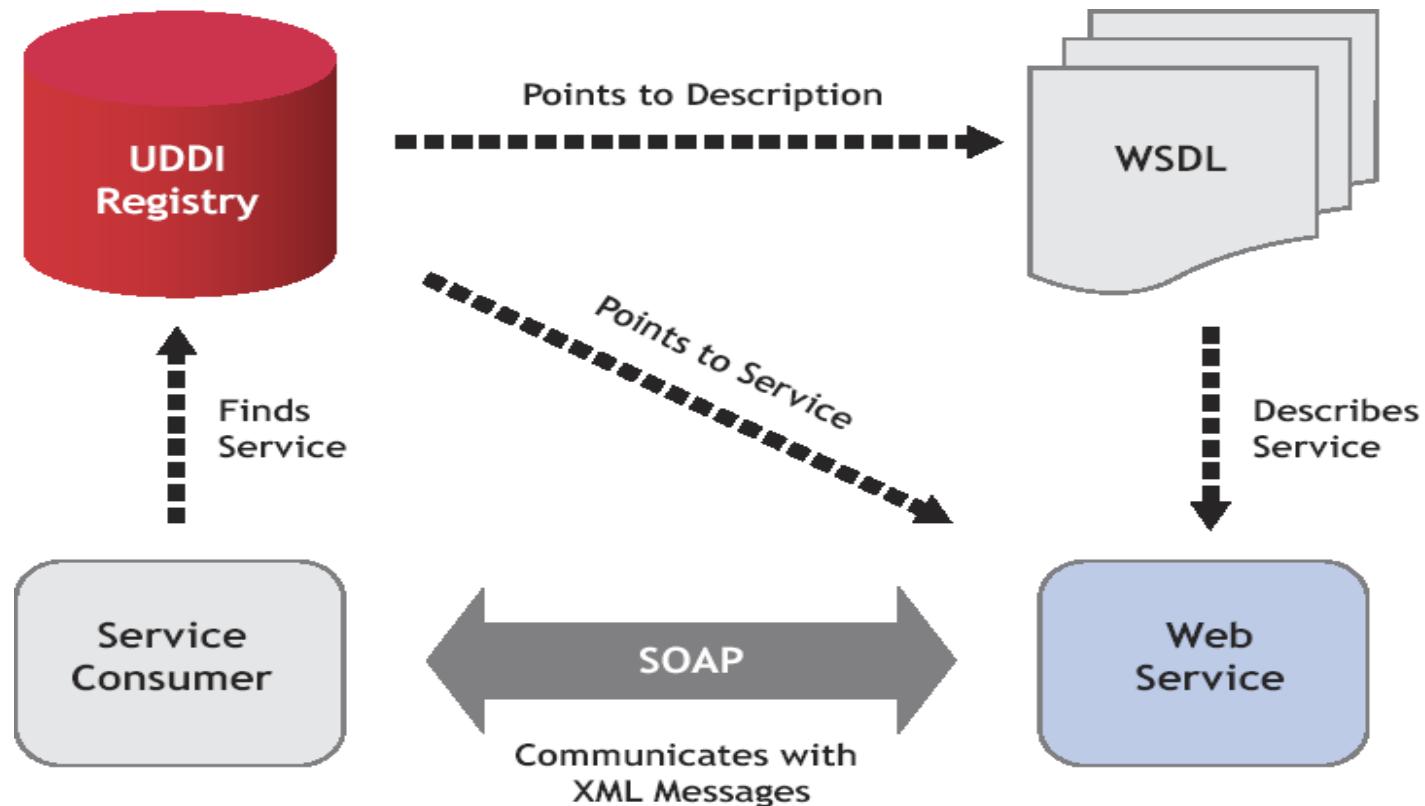
Semantic Web Vision



Web Services

- Web Services provide data and service.
- With standard web protocol (HTTP, HTML, XML, and SOAP) possible to connect, without knowing the concrete implementation.

Web Services



UDDI - Universal Description, Discovery, and Integration

[Stollberg et al., 05]

Web Services

- Technologies provide possibility to use WebServices
- but:
 - Only syntactic description available
 - Can be searched only based on syntax, using is manual
 - No semantic information
 - Not supporting semantic web

[Stollberg et al., 05]

Semantic Web Services

- Let's define a Web Service description language (**Web Service Description Ontologies**)
- Support ontologies, so that machines will be able to interpret (**Semantic Web aspect**)
- Let's define systems working on semantic systems, that are able to chose the correct services (**Web Service aspect**)

Semantic Web Services

- Automatic discovery

Find a book selling service

- Automatic parameters

Purchase the latest Delia Smith book

- Automatic composition and interoperation

Purchase the cheapest latest Delia Smith book

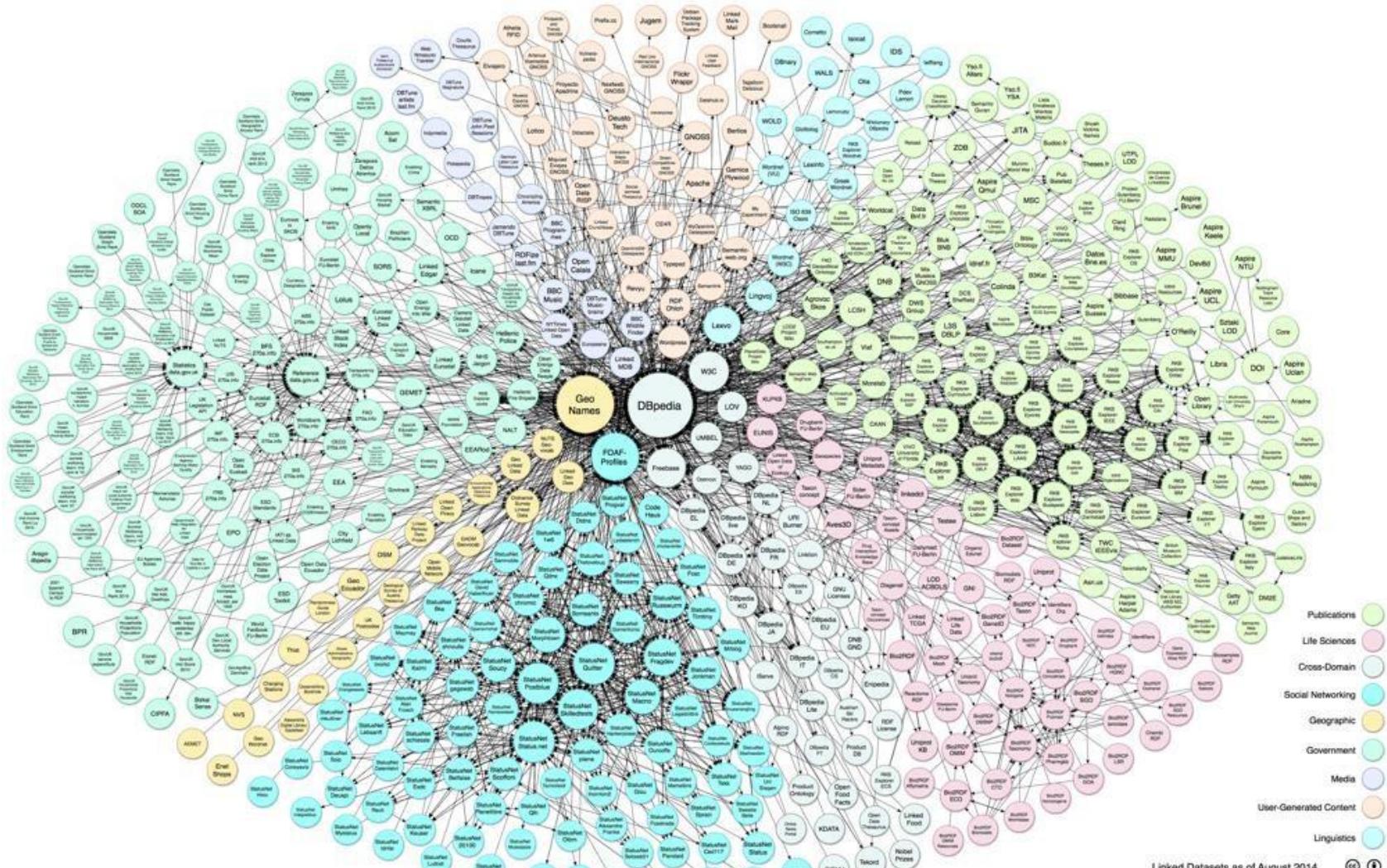
- Automatic execution monitoring

What is the status of my book order?

Linked Open Data

- Data are available and can be connected
- Semantic data can be reached typically on SPARQL Endpoints

LOD Cloud



DBpedia

- Semantic Wikipedia

DBpedia



About: Loránd Eötvös

An Entity of Type : [scientist](#), from Named Graph : <http://live.dbpedia.org>, within Data Space : live.dbpedia.org

Baron Loránd Eötvös de Vásárosnamény (27 July 1848 – 8 April 1919), more commonly called Baron Roland von Eötvös in English literature, was a Hungarian physicist. He is remembered today largely for his work on gravitation and surface tension.

Property	Value
dbpedia-owl:abstract	<ul style="list-style-type: none">Baron Loránd Eötvös de Vásárosnamény (27 July 1848 – 8 April 1919), more commonly called Baron Roland von Eötvös in English literature, was a Hungarian physicist. He is remembered today largely for his work on gravitation and surface tension.
dbpedia-owl:birthDate	<ul style="list-style-type: none">1848-07-27 (xsd:date)
dbpedia-owl:birthPlace	<ul style="list-style-type: none">dbpedia:Buda
dbpedia-owl:deathDate	<ul style="list-style-type: none">1919-04-08 (xsd:date)
dbpedia-owl:deathPlace	<ul style="list-style-type: none">dbpedia:Budapest
dbpedia-owl:field	<ul style="list-style-type: none">dbpedia:Physics
dbpedia-owl:knownFor	<ul style="list-style-type: none">dbpedia:Eötvös_ruledbpedia:Eötvös_experiment
dbpedia-owl:nationality	<ul style="list-style-type: none">dbpedia:Hungary
dbpedia-owl:thumbnail	<ul style="list-style-type: none">http://upload.wikimedia.org/wikipedia/commons/thumb/4/4b/Roland_Eotvos.jpg/200px-Roland_Eotvos.jpg
dbpedia-owl:wikiPageExternalLink	<ul style="list-style-type: none">http://www.mek.iif.hu/porta/szint/tarsad/tudtan/eotvos/html/stepcikk.htmlhttp://www.ncbi.nlm.nih.gov/pubmed/4932574http://www.elgi.hu/cgi-bin/cnt_eng
dbpprop:after	<ul style="list-style-type: none">dbpedia:Gyula_Wlassics
dbpprop:before	<ul style="list-style-type: none">dbpedia:Albin_Csáky
dbpprop:birthDate	<ul style="list-style-type: none">27 (xsd:integer)
dbpprop:birthPlace	<ul style="list-style-type: none">dbpedia:Buda
dbpprop:deathDate	<ul style="list-style-type: none">8 (xsd:integer)
dbpprop:deathPlace	<ul style="list-style-type: none">dbpedia:Budapest
dbpprop:field	<ul style="list-style-type: none">Physics
dbpprop:knownFor	<ul style="list-style-type: none">Eötvös ruleEötvös experiment
dbpprop:nationality	<ul style="list-style-type: none">Hungary
dbpprop:thumbnail	<ul style="list-style-type: none">http://upload.wikimedia.org/wikipedia/commons/thumb/4/4b/Roland_Eotvos.jpg/200px-Roland_Eotvos.jpg
dbpprop:wikiPageExternalLink	<ul style="list-style-type: none">http://www.mek.iif.hu/porta/szint/tarsad/tudtan/eotvos/html/stepcikk.htmlhttp://www.ncbi.nlm.nih.gov/pubmed/4932574http://www.elgi.hu/cgi-bin/cnt_eng

Semantic web

Federated queries

- Querying multiple endpoint in single queries
- We find information about certain entities on multiple endpoints

Query 3: Example of Federated SPARQL Query in the SPARQL 1.1

```
SELECT ?drugname ?indication
WHERE {
SERVICE <http://dbpedia.org/sparql>
{
    ?drug a dbpedia-owl:Drug .
    ?drug rdfs:label ?drugname .
    ?drug owl:sameAs ?drugbank .
}
SERVICE <http://www4.wiwiss.fu-berlin.de/drugbank/sparql>
{
?drugbank drugbank:indication ?indication .
}
}
```

Federated system

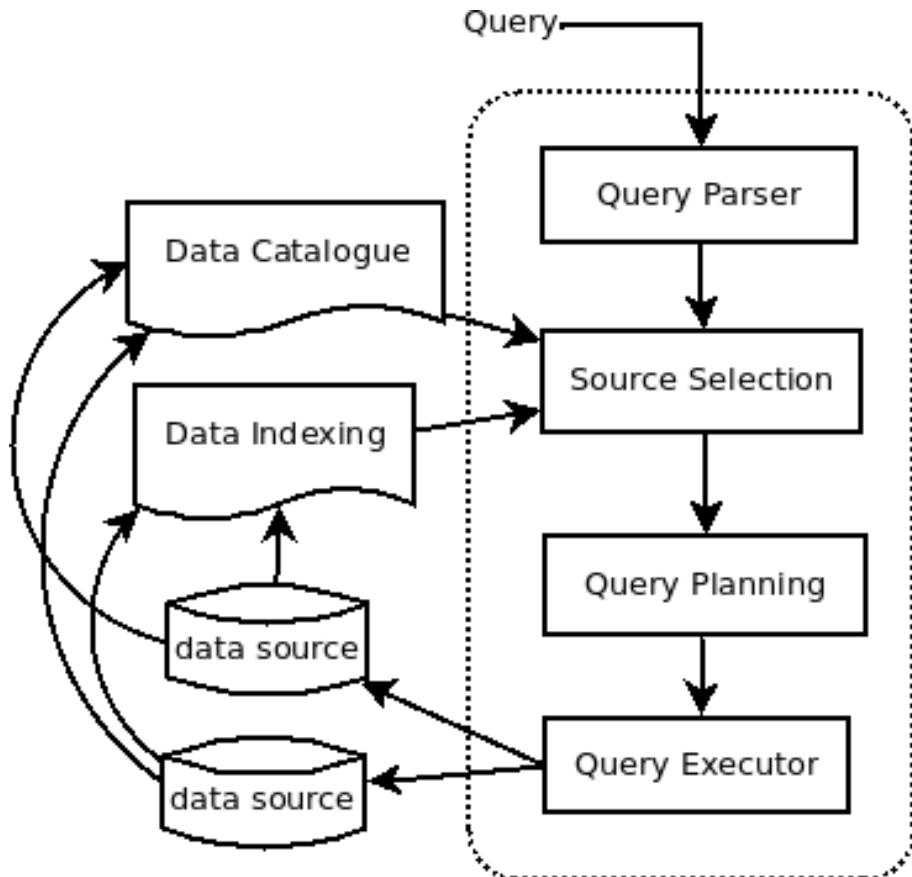
- Let's simplify the query

Query 4: Example of Federation SPARQL Query in the SPARQL 1.0 without SPARQL Endpoint specified

```
SELECT ?drugname ?indication  
WHERE {  
?drug a dbpedia-owl:Drug .  
?drug rdfs:label ?drugname .  
?drug owl:sameAs ?drugbank .  
?drugbank drugbank:indication ?indication .  
}
```

Federated system

1. Query parser
2. Source selection
 - a. ASK SPARQL Query
 - b. Data Catalogue
 - c. Data index
 - d. Caching



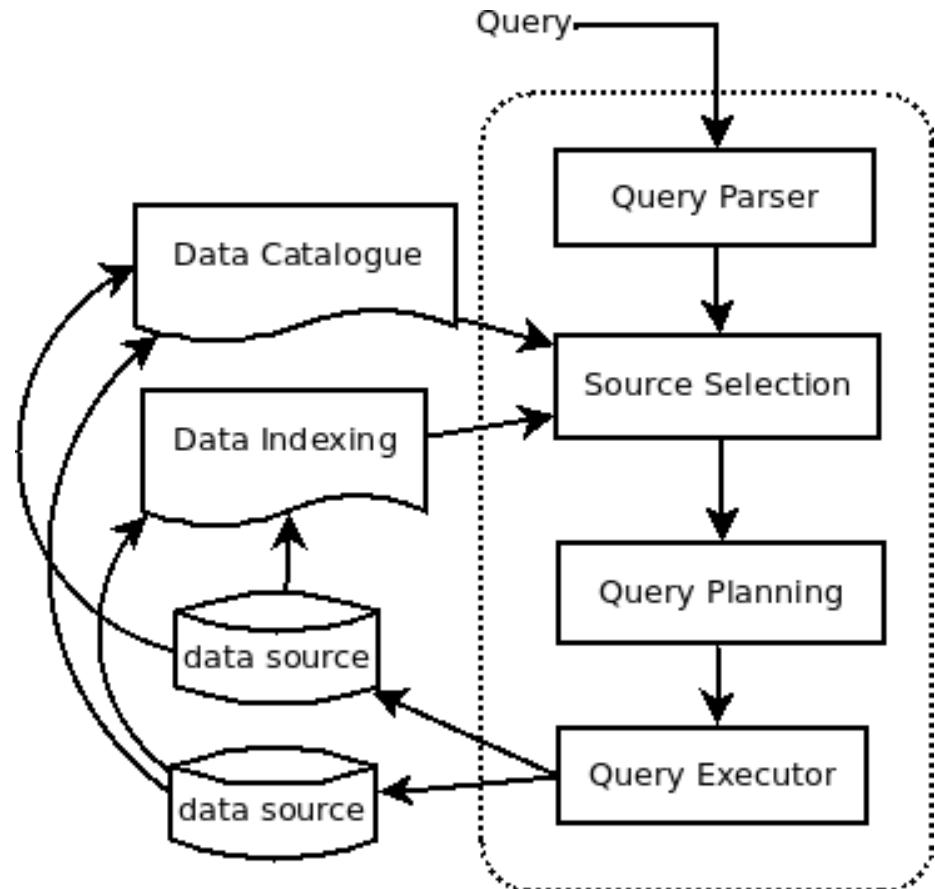
Federated system

3. Query planning

a. Subqueries

4. Execution

3. Join strategy



Applications

- Semantic browsers
- Thematic applications
- Search engines

Thematic browsers

http://www.w3.org/People/Berners-Lee/card#i Open



Tim Berners-Lee

http://www.w3.org/1999/02/22-owl-similes-owl#type

- Person 
- http://www.w3.org/2000/10/swap/persistence#

label

- Tim Berners-Lee 

sameAs

- Tim Berners-Lee [also at www.w3c.tu-berlin.de](http://www.w3c.tu-berlin.de) 

image



WebLinks

name

- Tim Berners-Lee 
- Timothy Berners-Lee 
- Tim Berners Lee 

Given name

- Timothy 

family name

- Berners-Lee 

sha1sum of a personal mailbox URI name

- 985c47c5a70db7407210ce0e4e65374a52605c 

workplace homepage

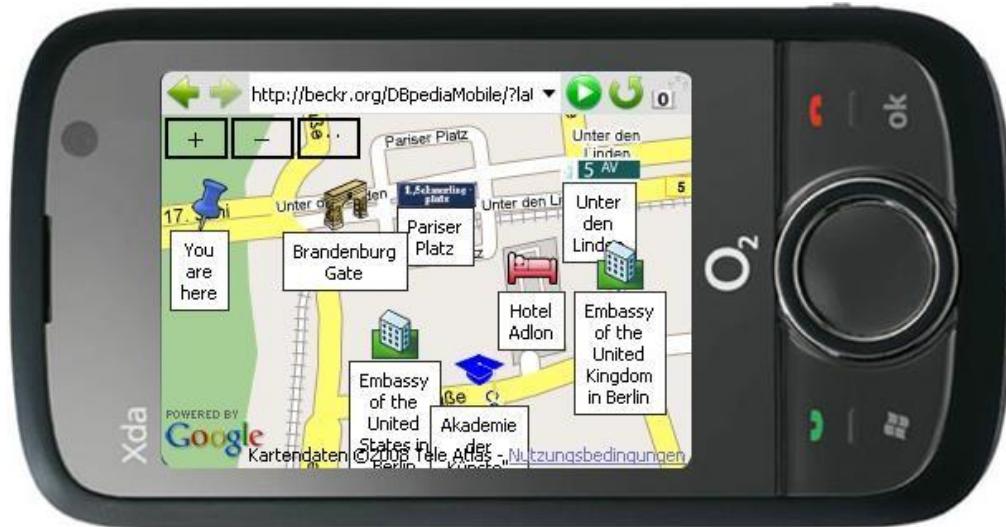
- http://www.w3.org/ 

nickname

- TimBL 
- TimBL 
- TimBL 

Theme specific applications

- Dbpedia mobile



Search engines

YAHOO! RESEARCH micro SEARCH ivan herman Examples: [1](#), [2](#), [3](#), [4](#). Note: response times may vary between 2-15 seconds. [About Feedback](#) (ver. 1.04)

Persons: 3 vCards: 35 Events: 25 Unfinished: 0 | 1 - 10 of about 224407 for ivan herman - 48 sec. ([About this page](#))

Search Results

1. [Ivan's private site](#)
<http://www.ivan-herman.net/> - 46k [Update metadata](#)
2. [Ivan Herman](#)

name: Ivan Herman
personal mailbox: <mailto:ivan@w3.org>
homepage: <http://www.ivan-herman.net>
Ivan Herman gives a talk on behalf of the China Office entitled "What is the ... Ben Adida, Elias Torres, and Ivan Herman give a tutorial entitled "RDFa: ...
<http://www.w3.org/People/Ivan/> - 25k [Update metadata](#)
homepage
[Ivan's private site](#)
<http://www.ivan-herman.net/> - 46k [Update metadata](#)
3. [Amsterdam, the Netherlands \(the city where I live...\)](#)
Lots of pictures taken when walking around this very ... Total images: 79 | Last update: 2008-02-16 | © JAlbum & Chameleon | Help © Ivan Herman, 2007 ...
<http://www.ivan-herman.net/Photos/Amsterdam/> - 53k [Update metadata](#)
4. [Ivan Herman](#)


A world map showing search results for "ivan herman". Callout boxes highlight specific locations: "Fairmont San Jose" in North America, "Ivan Herman" in Europe, and "Gandon" also in Europe. The map includes labels for continents (Europe, Asia, Africa, South America), oceans (North Pacific Ocean, Kara Sea, Laptev Sea), and various cities. A scale bar at the bottom left indicates distances up to 5000km/mi.

Search engines

tim berners-lee született

Internet Képek Térkép Videók Egyebek ▾ Keresőeszközök

Nagyjából 12 600 találat (0,37 másodperc)

1955. június 8. (életkor 58), London, Egyesült Királyság

Tim Berners-Lee, Született

Visszajelzés / További információ

[Tim Berners-Lee - Wikipédia](#)
hu.wikipedia.org/wiki/Tim_Berners-Lee ▾

Sir Timothy John "Tim" Berners-Lee, KBE, (TimBL vagy TBL) (sz. London, 1955. június 8.) a Világháló (World Wide Web) pontosabban a HTML, a HTTP és több ...

[BLOG és BLOGGER – egy új kultúra születése | Food & Wine](#)
www.foodandwine.hu/2011/.../blog-es-blogger-egy-uj-kultura-szuletese/ ▾

2011.07.14. - Jorn Barger 1953-ban, Yellow Springsben, Ohioban született. ... Közülük az egyik, Sir Timothy John "Tim" Berners-Lee, az internetes világ nagy ...

[SG.hu - 50 éves a web atya](#)



További képek

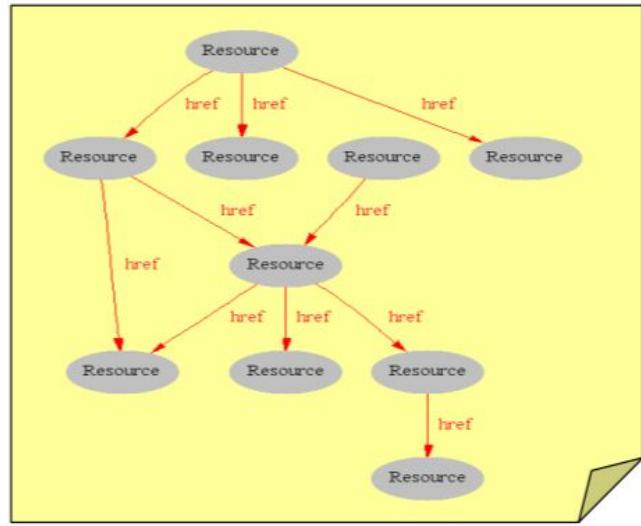
Tim Berners-Lee

Sir Timothy John "Tim" Berners-Lee, KBE, a Világháló pontosabban a HTML, a HTTP és több hasonló protokoll kifejlesztője és a World Wide Web Consortium vezetője, azé a cége, mely a Web további fejlődését irányítja. [Wikipédia](#)

Született: 1955. június 8. (életkor 58), London, Egyesült Királyság
Szülők: Mary Lee Woods, Conway Berners-Lee

SemWeb - Original Page 7 - SemWeb - Appendix
 Resource - href

Syntactic Web



Data models and databases

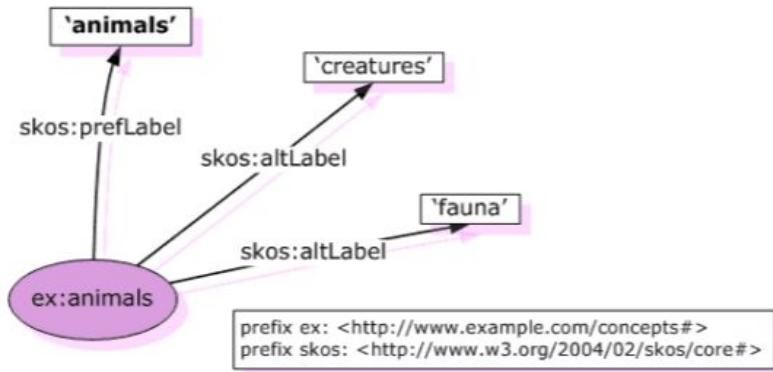
Semantic web

[Handler & Miletić 02]

SemWeb - Original Page 21 - SemWeb - Appendix
RDF Example

Ex: animals -> skos: prefLabel -> 'animals'
-> skos: altLabel -> 'creatures'
-> skos: altLabel -> 'fauna'

RDF Example



```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:skos="http://www.w3.org/2004/02/skos/core#">

  <skos:Concept rdf:about="http://www.example.com/concepts#animals">
    <skos:prefLabel>animals</skos:prefLabel>
    <skos:altLabel>creatures</skos:altLabel>
    <skos:altLabel>fauna</skos:altLabel>
  </skos:Concept>

</rdf:RDF>
```

Data models and databases

Source: <http://www.w3.org/TR/swbp-skos-core-guide/>

25

SemWeb - Original Page 23 - SemWeb - Appendix

RDF Formats

Feature Expresses RDF 1.0 @prefix collections numeric literals literal subj RDF Path Rules

Syntax Ntriples Turtle N3 RDF N3 Rules N3 y y plus prime forAll x y z a <>

RDF Formats

Feature	Expresses RDF 1.0	@prefix [] ; a	Collections	Numeric literals	Literal subj	RDF Path	Rules	Formulae
syntax:			(<a>)	2	7 a n:prime	xly^z	{?x}=>{?x}	{ } @forAll
NTriples	y							
Turtle	y	y	y					
N3 RDF	y	y	y	y	y	y		
N3 Rules	y plus	y	y	y	y	y	y	
N3	y plus	y	y	y	y	y	y	y

Types of Ontologies

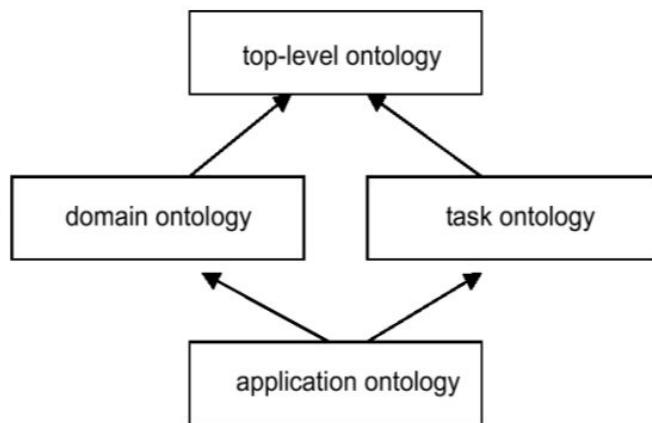
Top-level ontology, domain ontology, task ontology, application ontology

Types of Ontologies

[Guarino, 98]

Describe **very general concepts** like space, time, event, which are independent of a particular problem or domain. It seems reasonable to have unified top-level ontologies for large communities of users.

Describe the vocabulary related to a **generic domain** by specializing the concepts introduced in the top-level ontology.



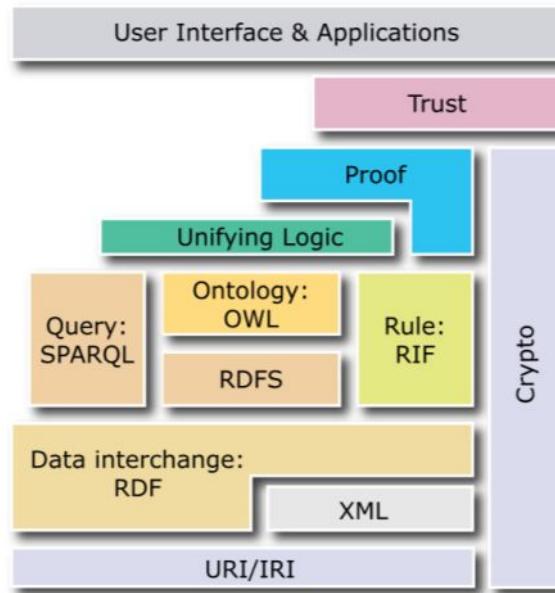
Describe the vocabulary related to a **generic task or activity** by specializing the top-level ontologies.

These are the most specific ontologies. Concepts in application ontologies often correspond to **roles played by domain entities while performing a certain activity**.

Semantic Web Vision

User interface applications, trust, proof, unifying logic, crypto, query: sparql, Ontology: OWL, Rule: RIF, RDFS, Data interchange: RDF, XML, URI/IRI

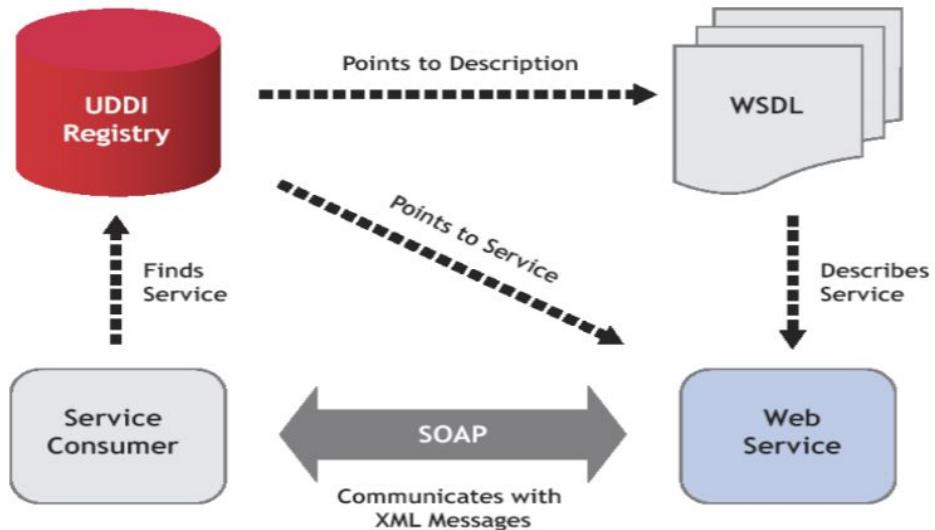
Semantic Web Vision



Web Services

UDDI Registry, Points to Description, WSDL, Describes Service, Web Service, SOAP, Communicates with XML Messages, Service Customer, Finds Service, Points to Service, UDDI - Universal Description, Discovery, and Integration - Stollberg et al., 05

Web Services



UDDI - Universal Description, Discovery, and Integration

[Stollberg et al., 05]

Federated queries • Querying multiple endpoint in single queries • We find information about certain entities on multiple endpoints

Query 3: Example of Federated SPARQL Query in the SPARQL 1.1
SELECT ?drugname ?indication
WHERE {
SERVICE <http://dbpedia.org/sparql>
{
?drug a dbpedia-owl:Drug .
?drug rdfs:label ?drugname .
?drug owl:sameAs ?drugbank .
}
SERVICE <http://www4.wiwiss.fu-berlin.de/drugbank/sparql>
{
?drugbank drugbank:indication ?indication .
}
}

Federated queries

- Querying multiple endpoint in single queries
- We find information about certain entities on multiple endpoints

Query 3: Example of Federated SPARQL Query in the SPARQL 1.1

```
SELECT ?drugname ?indication
WHERE {
SERVICE <http://dbpedia.org/sparql>
{
?drug a dbpedia-owl:Drug .
?drug rdfs:label ?drugname .
?drug owl:sameAs ?drugbank .
}
SERVICE <http://www4.wiwiss.fu-berlin.de/drugbank/sparql>
{
?drugbank drugbank:indication ?indication .
}
}
```

Federated system • Let's simplify the query

Query 4: Example of Federation SPARQL Query in the SPARQL 1.0 without SPARQL Endpoint specified

Federated system

- Let's simplify the query

Query 4: Example of Federation SPARQL Query in the SPARQL 1.0 without SPARQL Endpoint specified

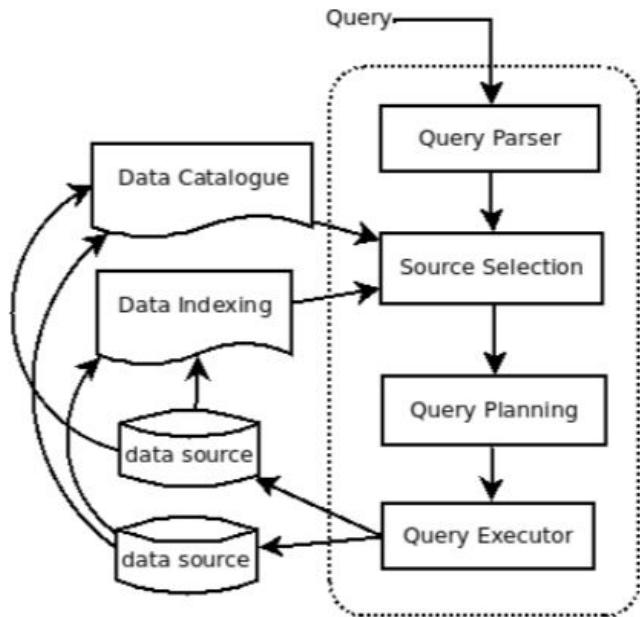
```
SELECT ?drugname ?indication
WHERE {
?drug a dbpedia-owl:Drug .
?drug rdfs:label ?drugname .
?drug owl:sameAs ?drugbank .
?drugbank drugbank:indication ?indication .
}
```

Federated system

Query Query Parser Source Selection Query Planning Query Executor data source data indexing data catalogue

Federated system

1. Query parser
2. Source selection
 - a. ASK SPARQL Query
 - b. Data Catalogue
 - c. Data index
 - d. Caching



Federated system

Query Query Parser Source Selection Query Planning Query Executor data source data indexing data catalogue

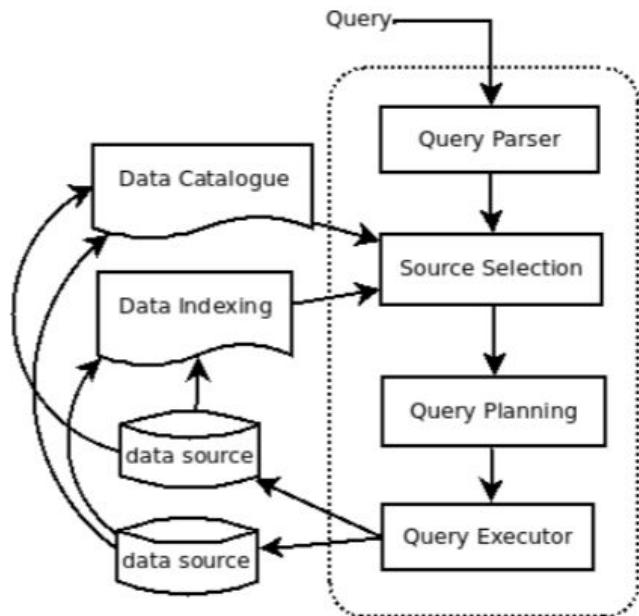
Federated system

3. Query planning

a. Subqueries

4. Execution

3. Join strategy



Design of distributed databases

Lehotay-Kéry Péter

Based on

Lecture of Kiss Attila

Architecture

- Defines the structure of the system
 - components identified
 - functions of each component defined
 - interrelationships and interactions between components defined

Standardization

Reference Model

- A conceptual framework whose purpose is to divide standardization work into manageable pieces and to show at a general level how these pieces are related to one another.

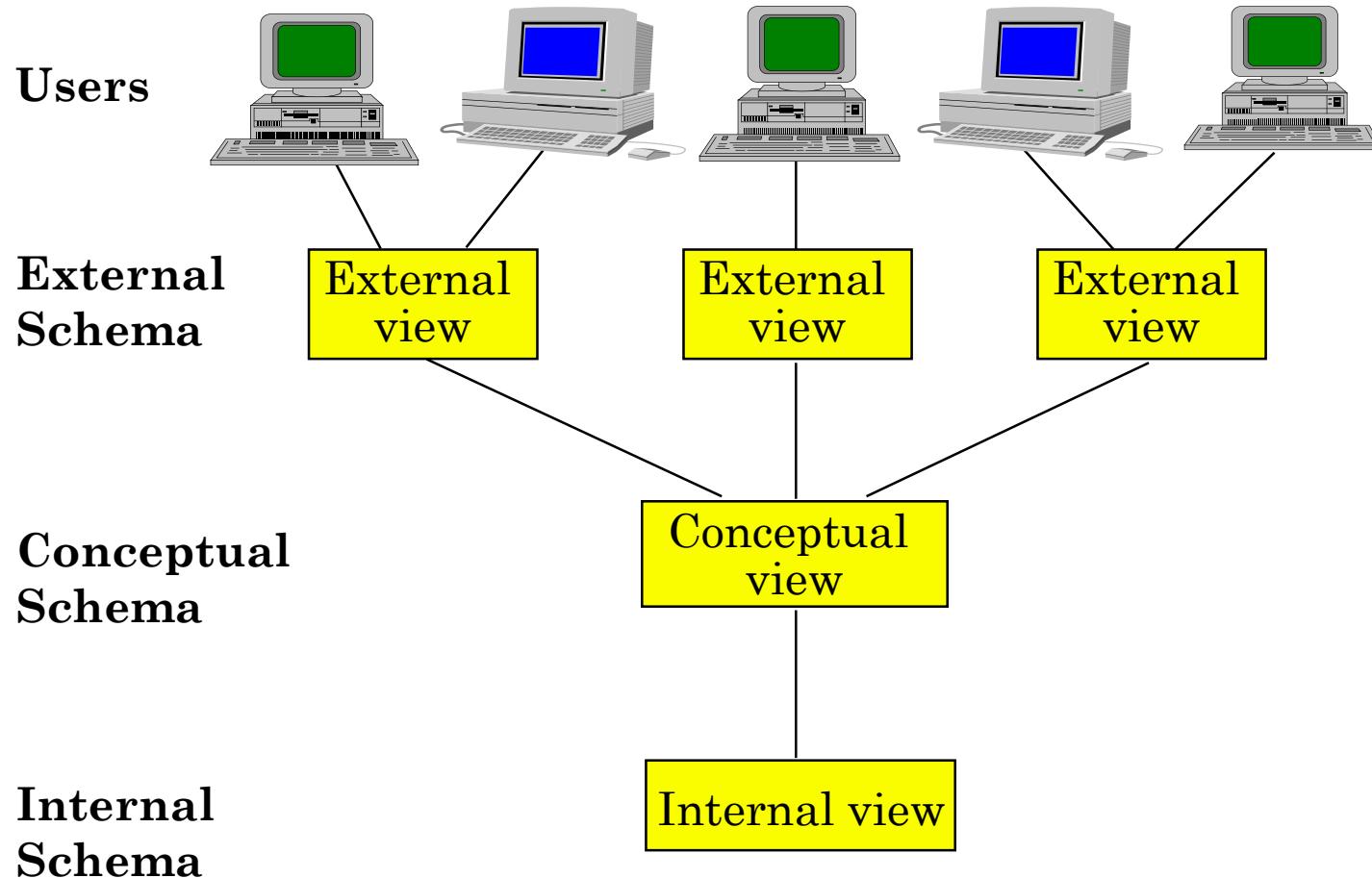
Approaches

- **Component-based**
 - Components of the system are defined together with the interrelationships between components.
 - Good for design and implementation of the system.
- **Function-based**
 - Classes of users are identified together with the functionality that the system will provide for each class.
 - The objectives of the system are clearly identified. But how do you achieve these objectives?
- **Data-based**
 - Identify the different types of describing data and specify the functional units that will realize and/or use data according to these views.

ANSI/SPARC Architecture

- Data-based approach to plan a DBMS.
- The essence of the model is the 3 kind of representation of the data
- On the different levels we get the schemas corresponding to the levels with collections of views. The connections between the views are given by transformations.
- Distinguishing the External and Conceptual views give the logical data independency
- Distinguishing the Conceptual and Internal views give the physical data independency

ANSI/SPARC Architecture



ANSI/SPARC Architecture

- Conceptional schema: Abstract representation of the database where the data and context described independently from the applications and psysical implementation.
- Internal schema: on the lowest level of the archirecure, representing the physical data description
- External view: Users can view the database corresponding to this.

Conceptual Schema Definition

```
RELATION EMP [
    KEY = {ENO}
    ATTRIBUTES = {
        ENO      : CHARACTER(9)
        ENAME   : CHARACTER(15)
        TITLE    : CHARACTER(10)
    }
]
RELATION PAY [
    KEY = {TITLE}
    ATTRIBUTES = {
        TITLE    : CHARACTER(10)
        SAL     : NUMERIC(6)
    }
]
```

Conceptual Schema Definition

```
RELATION PROJ [
    KEY = {PNO}
    ATTRIBUTES = {
        PNO      : CHARACTER(7)
        PNAME   : CHARACTER(20)
        BUDGET  : NUMERIC(7)
    }
]
RELATION ASG [
    KEY = {ENO,PNO}
    ATTRIBUTES = {
        ENO      : CHARACTER(9)
        PNO      : CHARACTER(7)
        RESP     : CHARACTER(10)
        DUR      : NUMERIC(3)
    }
]
```

Internal Schema Definition

```
RELATION EMP [
    KEY = {ENO}
    ATTRIBUTES = {
        ENO      : CHARACTER(9)
        ENAME    : CHARACTER(15)
        TITLE    : CHARACTER(10)
    }
]
```



```
INTERNAL_REL EMPL [
    INDEX ON E# CALL EMINX
    FIELD = {
        HEADER    : BYTE(1)
        E#        : BYTE(9)
        ENAME     : BYTE(15)
        TIT       : BYTE(10)
    }
]
```

External View Definition – Example 1

Create a BUDGET view from the PROJ relation

```
CREATE      VIEW      BUDGET(PNAME, BUD)
AS          SELECT    PNAME, BUDGET
                  FROM    PROJ
```

External View Definition – Example 2

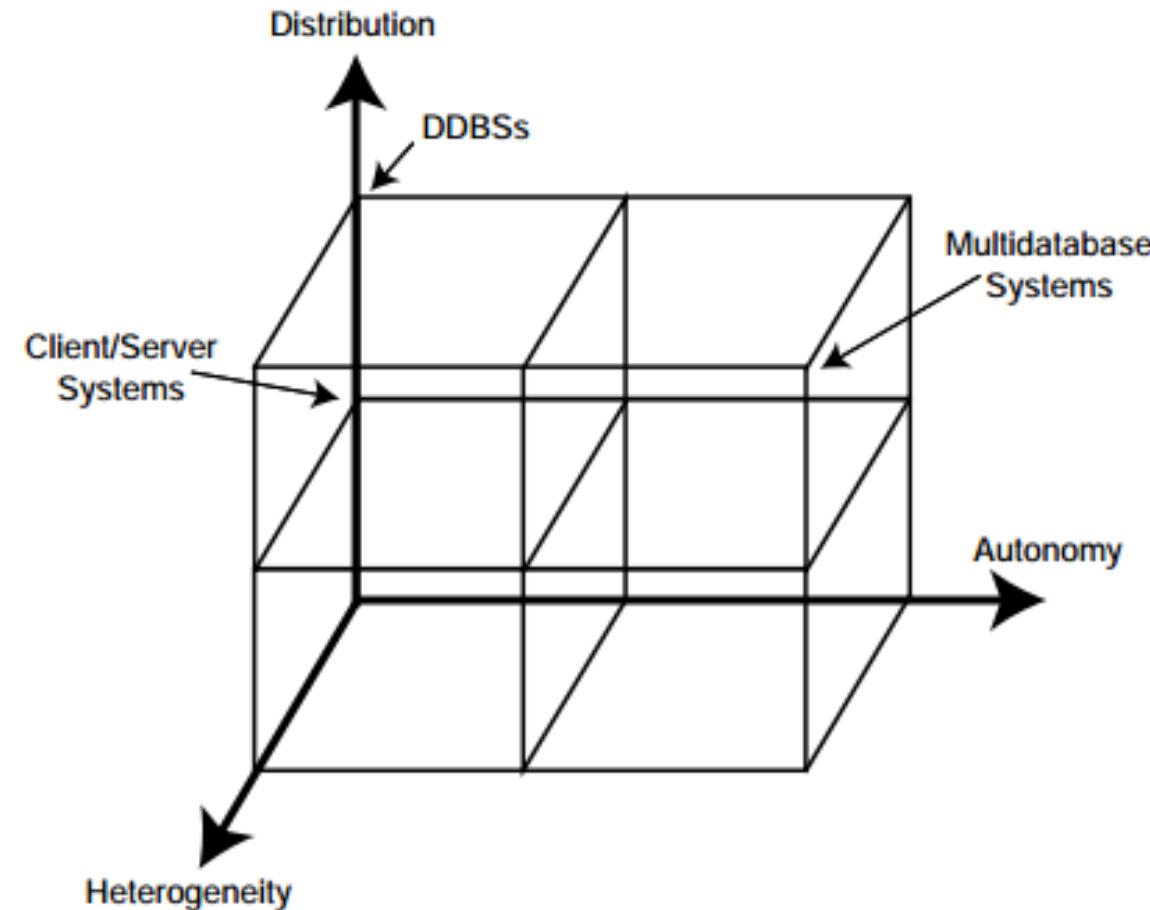
Create a Payroll view from relations EMP and
TITLE_SALARY

```
CREATE   VIEW    PAYROLL (ENO, ENAME, SAL)
AS        SELECT   EMP.ENO,EMP.ENAME,PAY.SAL
          FROM    EMP, PAY
          WHERE   EMP.TITLE = PAY.TITLE
```

Dimensions of the Problem

- Distribution
 - Whether the components of the system are located on the same machine or not
- Heterogeneity
 - Various levels (hardware, communications, operating system)
 - DBMS important one
 - data model, query language, transaction management algorithms
- Autonomy
 - Not well understood and most troublesome
 - Various versions
 - Design autonomy: Ability of a component DBMS to decide on issues related to its own design.
 - Communication autonomy: Ability of a component DBMS to decide whether and how to communicate with other DBMSs.
 - Execution autonomy: Ability of a component DBMS to execute local operations in any manner it wants to.

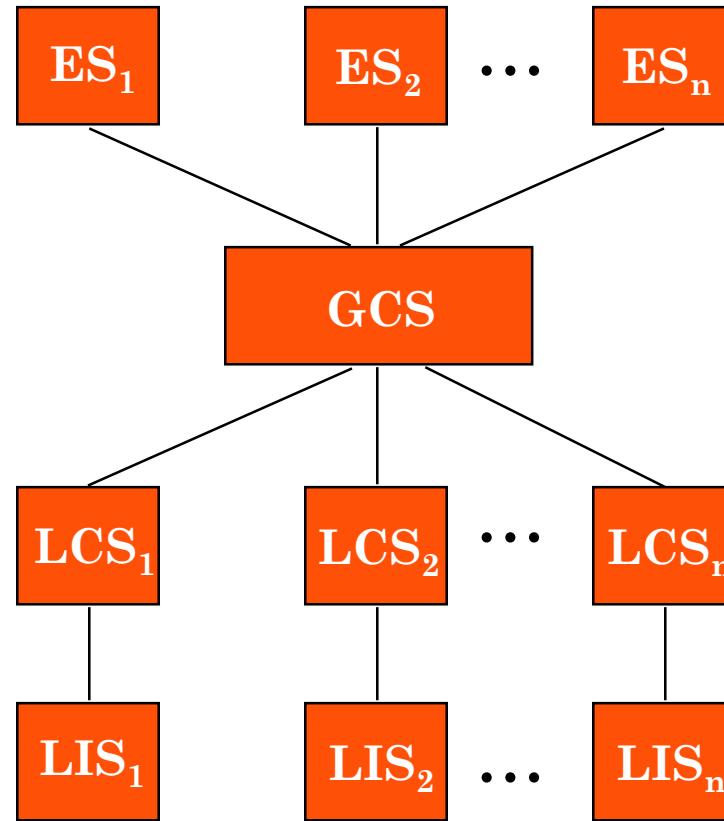
DBMS Implementation Alternatives



Datalogical Distributed DBMS Architecture (DDBS)

- Data layout can be different on the different machines, Local Inner Schema (LIS) is needed.
- A single, global conceptual schema is needed (GCS), this contains the logical structure of all data type.
- To handle the fragmentation and replication of data, Local Conceptual Schemas (LCS) are also needed, containing the more abstract, logical structure of the locally stored data types. The union of these is the GCS.
- Users can reach the database system through the External Schemas (ES).
- Global queries are compiled into multiple local queries by the DBMS, which are executed at the right place.

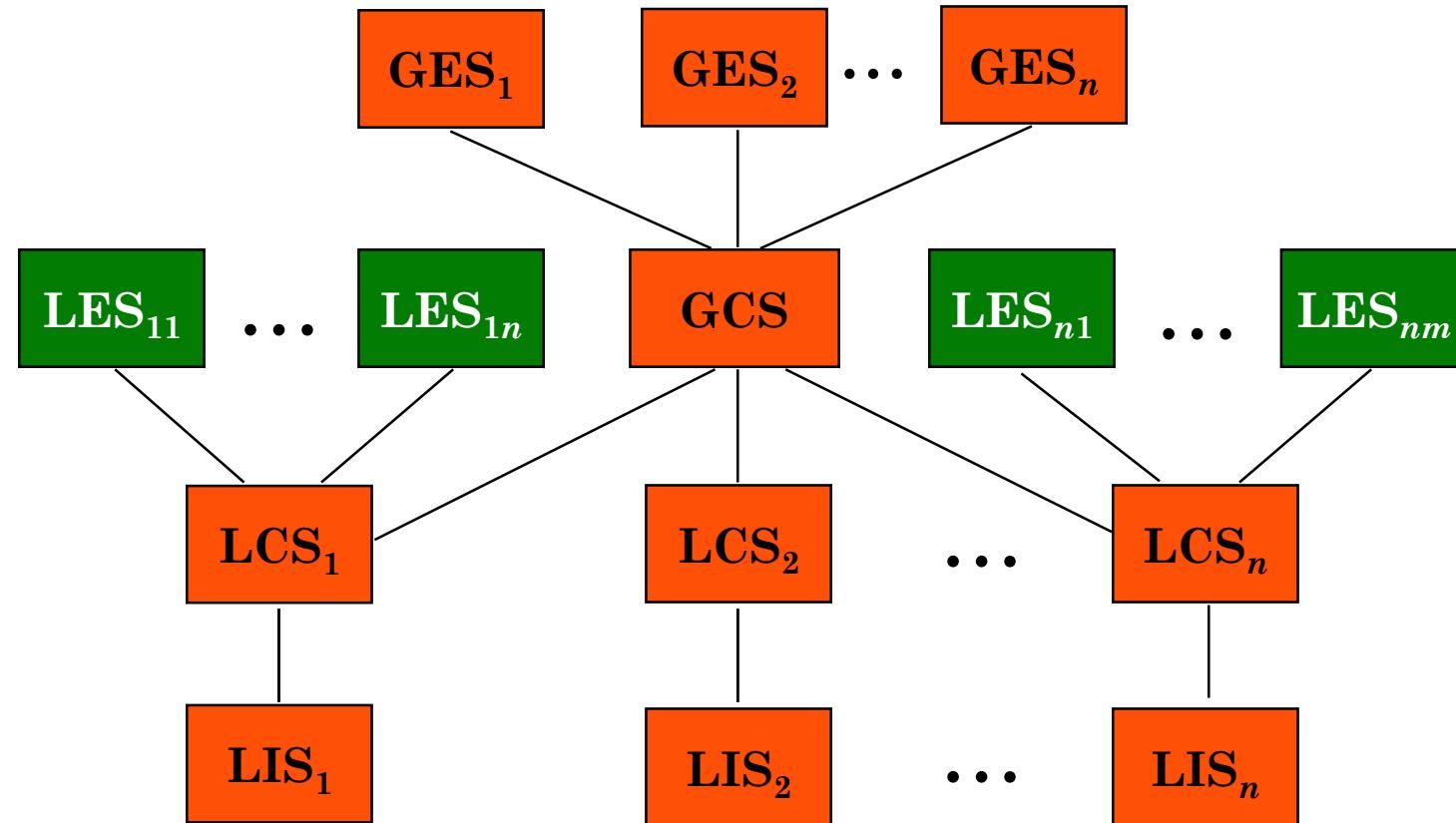
Datalogical Distributed DBMS Architecture (DDBS)



Multi-DBMS Architecture

- Fully autonomous
 - They do not know how to work together
 - Perhaps they do not know about the existence of each other
 - Perhaps they do not know how to communicate with each other
- GCS only describes those databases that the local DBMSs want to share. The shared data are described in the local export schema (LES) in each DBMSs.

Datalogical Multi-DBMS Architecture (MDBS)

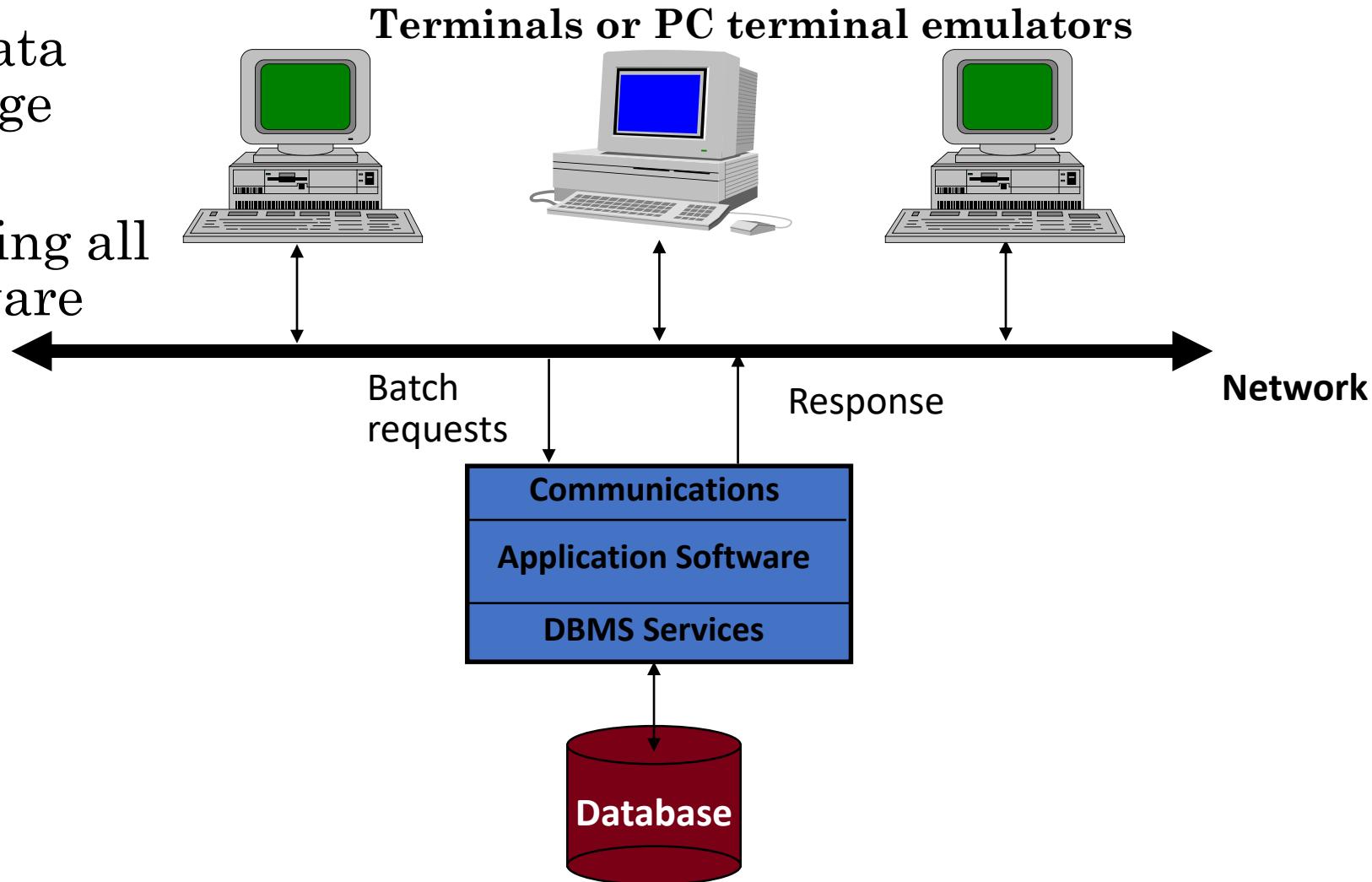


Timesharing Access to a Central Database

- The original idea is the user connects to the DBMS with time-sharing access.
- Then DBMS shares its processor time based on some scheduling between serving the users.

Timesharing Access to a Central Database

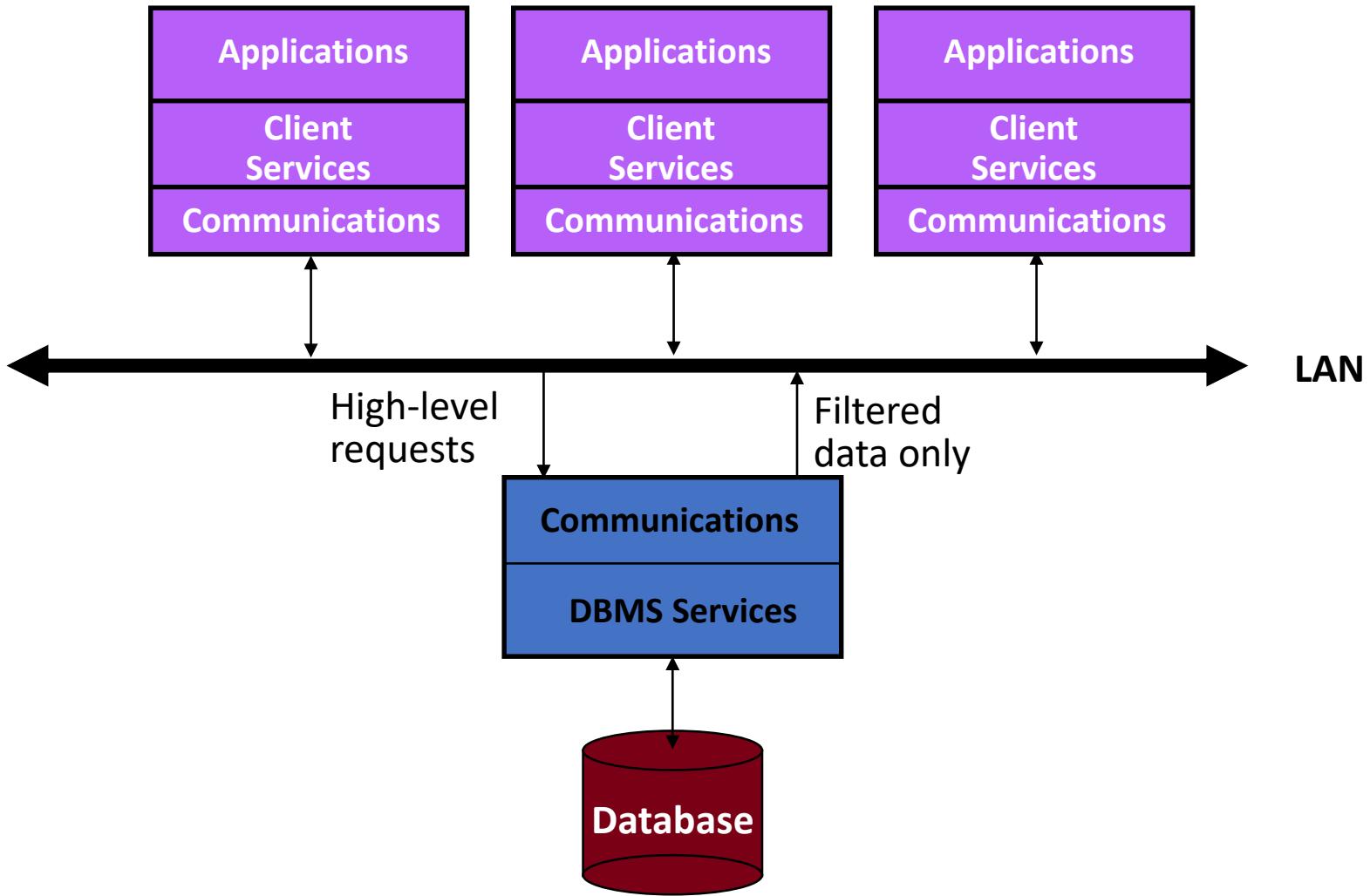
- No data storage
- Host running all software



Advantages of Client-Server Architectures

- More efficient division of labor
- Horizontal and vertical scaling of resources
- Better price/performance on client machines
- Ability to use familiar tools on client machines
- Client access to remote data (via standards)
- Full DBMS functionality provided to client workstations
- Overall better system price/performance

Multiple Clients/Single Server



Problems With Multiple-Client/Single Server

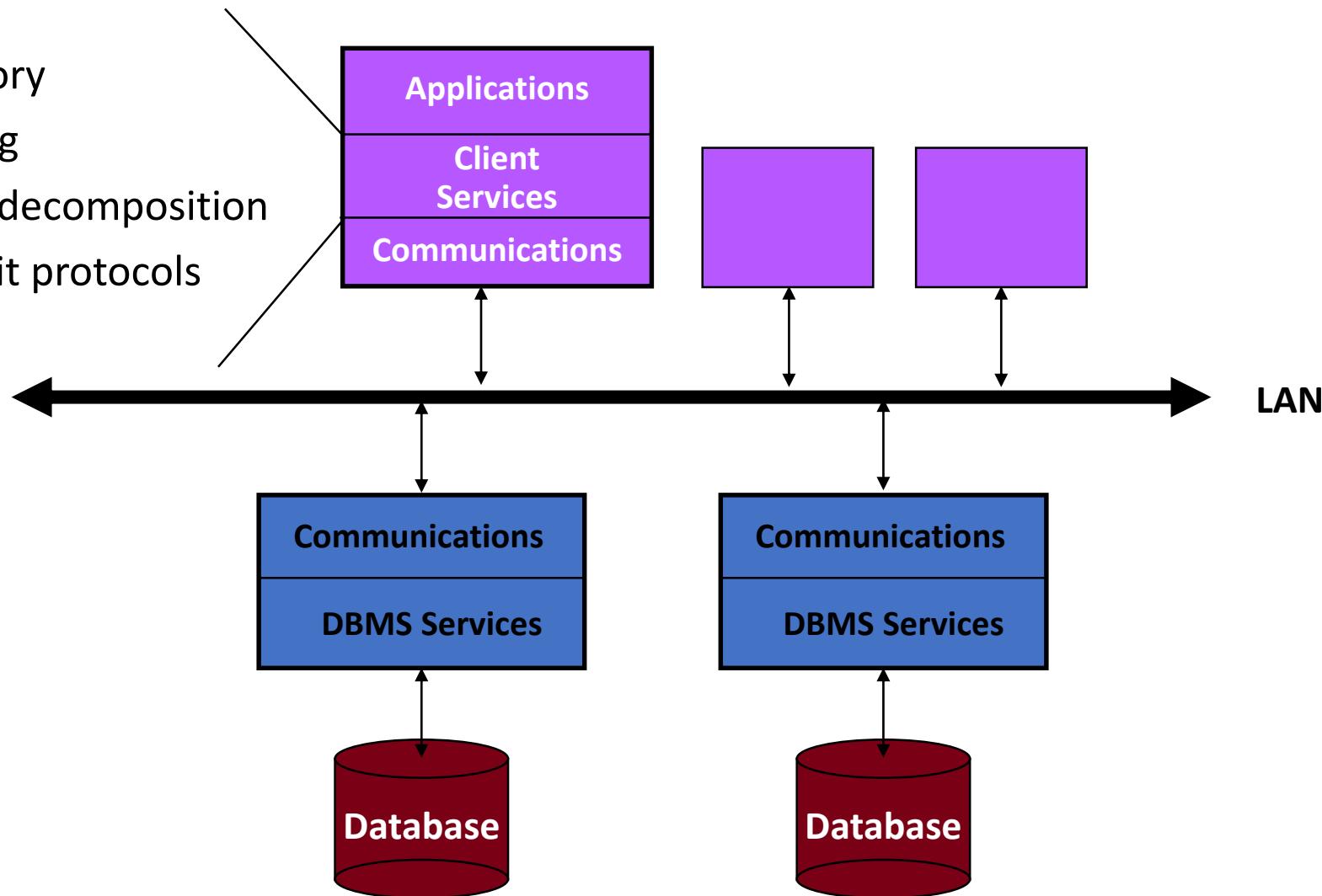
- Server forms bottleneck
- Server forms single point of failure
- Database scaling difficult

Multiple Clients/Multiple Servers

- Two type:
 - Every single client handles itself the connection to the corresponding server or servers. Code on server side is simpler but it leads to fat clients.
 - A client only has to communicate with its 'own' server, then further communications happens between the servers. This case leads to thin clients and servers execute most of the data handling.

Multiple Clients/Multiple Servers

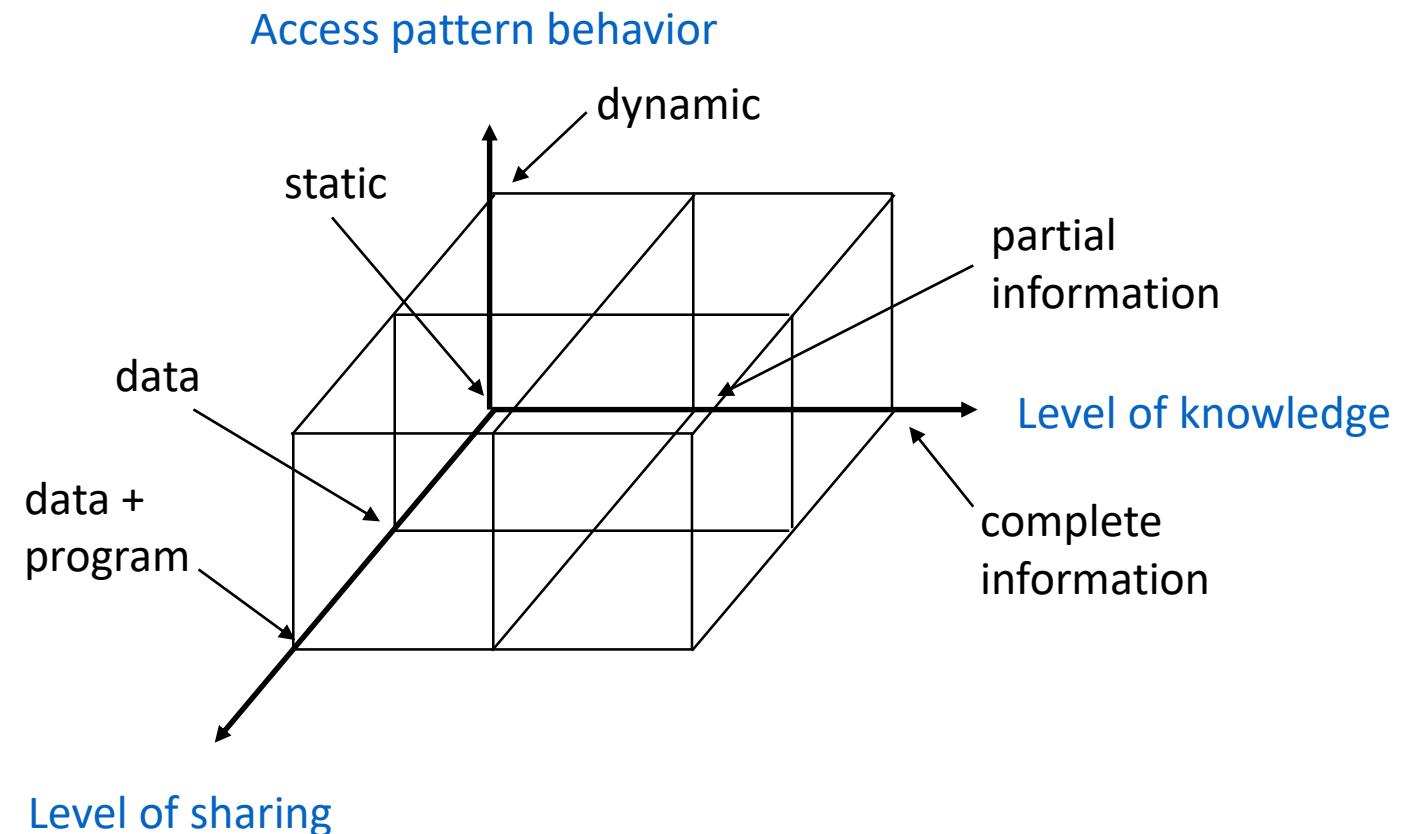
- directory
- caching
- query decomposition
- commit protocols



Dimensions of the Problem

- Level of sharing:
 - If no data shared, then the application and the corresponding data is executed on a single place.
 - Sharing only the data
 - Sharing the data and the programs
- Access pattern behavior: With more static access pattern behaviors, the easier to plan how to distribute everything ensuring data localization.
- Level of knowledge: Knowledge about the accessing models. If we possess the full knowledge, we would be able to predict with confidence all accesses with their nature in the future. With good chance we know what are the functionalities of the DBMS, when accessing it these will be used.

Dimensions of the Problem



Distribution Design Issues

- Why fragment at all?
- How to fragment?
- How much to fragment?
- How to test correctness?
- How to allocate?
- Information requirements?

Fragmentation Alternatives – Horizontal

PROJ_1 : projects with budgets less than \$200,000

PROJ_2 : projects with budgets greater than or equal to \$200,000

PROJ			
PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris
P5	CAD/CAM	500000	Boston

PROJ_1

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	15000	Montreal
P2	Database Develop	135000	New York

PROJ_2

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris
P5	CAD/CAM	500000	Boston

Fragmentation Alternatives – Vertical

PROJ_1 : information about project budgets

PROJ_2 : information about project names and locations

PROJ

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris
P5	CAD/CAM	500000	Boston

PROJ₁

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000
P5	500000

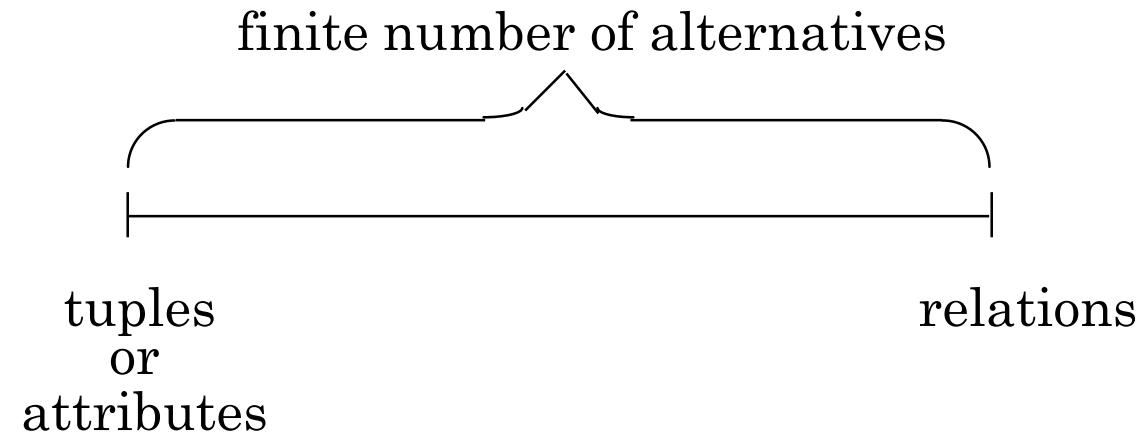
PROJ₂

PNO	PNAME	LOC
P1	Instrumentation	Montreal
P2	Database Develop	New York
P3	CAD/CAM	New York
P4	Maintenance	Paris
P5	CAD/CAM	Boston

Why fragment?

- We could distribute the relations redundant to the places where they needed. But this would be a waste of networking and memory resources.
- Not many applications need a whole data table, only a view on the relation needed.

Degree of Fragmentation



Finding the suitable level of partitioning
within this range

Correctness of Fragmentation

- Completeness
 - Decomposition of relation R into fragments R_1, R_2, \dots, R_n is complete if and only if each data item in R can also be found in some R_i
- Reconstruction
 - If relation R is decomposed into fragments R_1, R_2, \dots, R_n , then there should exist some relational operator ∇ such that
$$R = \nabla_{1 \leq i \leq n} R_i$$
- Disjointness
 - If relation R is decomposed into fragments R_1, R_2, \dots, R_n , and data item d_j is in R_j , then d_j should not be in any other fragment R_k ($k \neq j$).

Allocation Alternatives

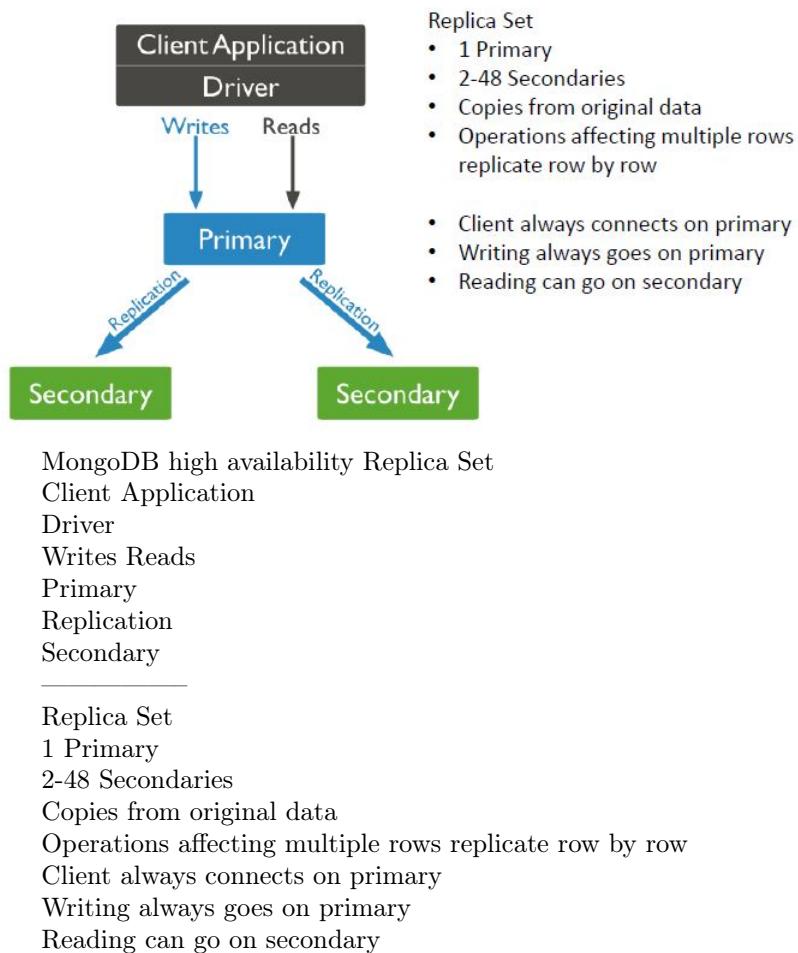
- Non-replicated
 - partitioned : each fragment resides at only one site
- Replicated
 - fully replicated : each fragment can be found at each site
 - partially replicated : each fragment can be found at some of the sites
- Rule of thumb:

If $\frac{\text{read - only queries}}{\text{update queries}} \geq 1$ replication is advantageous,
otherwise replication may cause problems

1 mongodb.pdf - Week 6 - NoSQL, MongoDB, Document Store

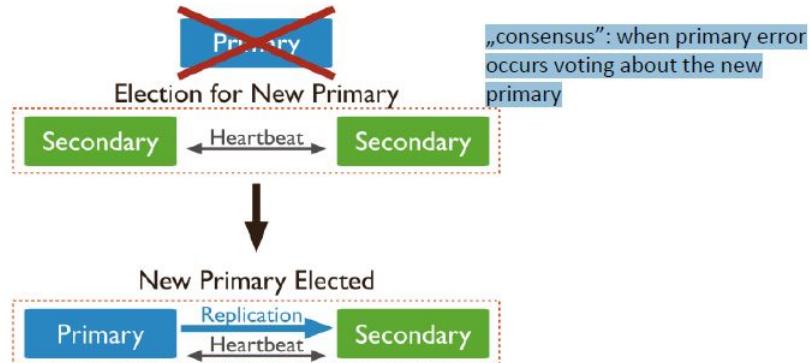
1.1 mongodb.pdf page 9 - Week 6 - NoSQL, MongoDB, Document Store

MongoDB high availability



1.2 mongodb.pdf page 10 - Week 6 - NoSQL, MongoDB, Document Store

MongoDB high availability



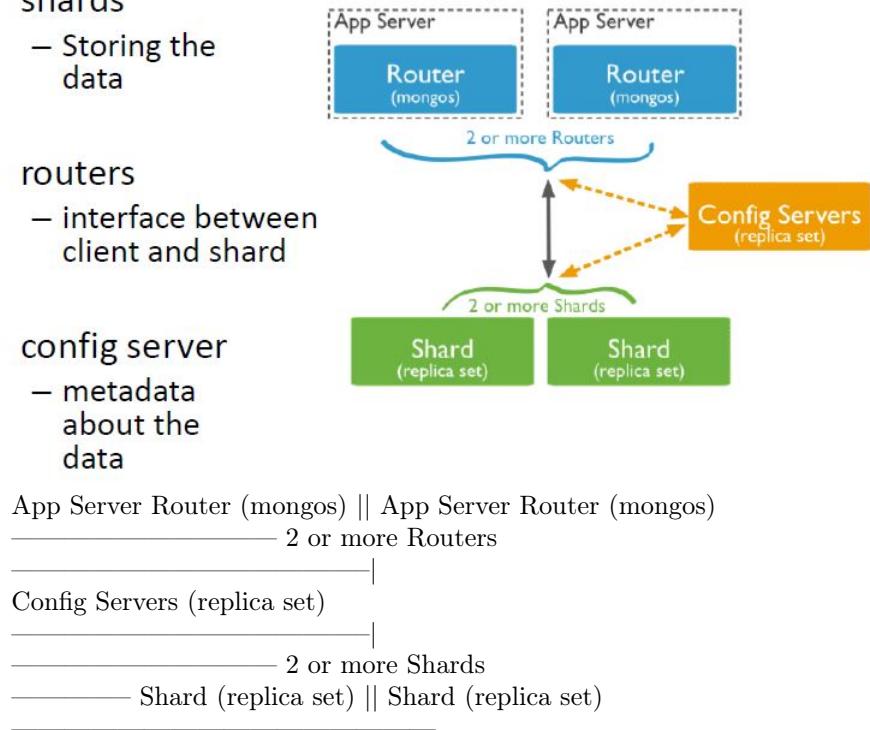
MongoDB high availability
Not primary
Election for New Primary
Secondary (Heartbeat) Secondary
New Primary Elected
Primary (Replication -> | <- Heartbeat) Secondary

consensus: when primary error occurs voting about the new primary

1.3 mongodb.pdf page 12 - Week 6 - NoSQL, MongoDB, Document Store

MongoDB sharding components

- shards
 - Storing the data
- routers
 - interface between client and shard
- config server
 - metadata about the data



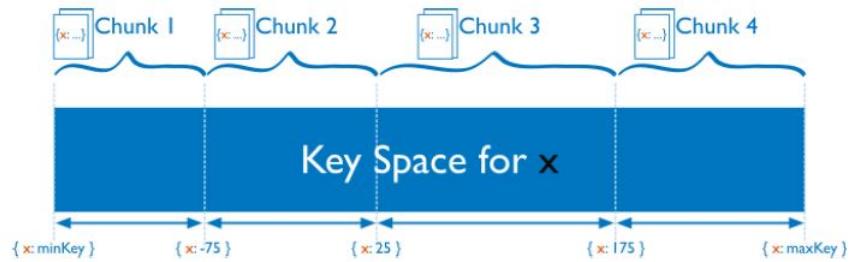
MongoDB sharding components
shards – Storing the data
routers – interface between client and shard
config server – metadata about the data

1.4 mongodb.pdf page 13 - Week 6 - NoSQL, MongoDB, Document Store

Sharding solutions

Prev

- Range based sharding
 - shard key storing based on domains
 - Efficient domain queries



Chunk 1 + Chunk 2 + Chunk 3 + Chunk 4 = Shard

Sharding solutions

Range based sharding

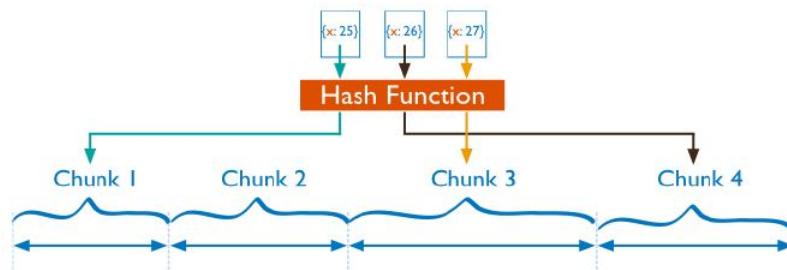
- shard key storing based on domains
- Efficient domain queries

1.5 mongodb.pdf page 14 - Week 6 - NoSQL, MongoDB, Document Store

Sharding solutions

Hash based sharding

- distributing data based on given hash function



key -> Hash Function { Chunk 1 + Chunk 2 + Chink 3 + Chunk 4 }

Sharding solutions

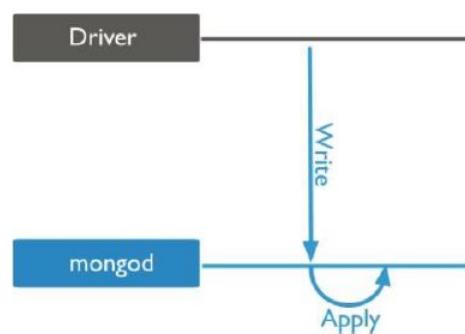
Hash based sharding

- distributing data based on given hash function

1.6 mongodb.pdf page 17 - Week 6 - NoSQL, MongoDB, Document Store

Unacknowledged

- Client does not get feedback of writing
- Very fast writing
- Possible data loss!
 - Networking error
 - Key collision

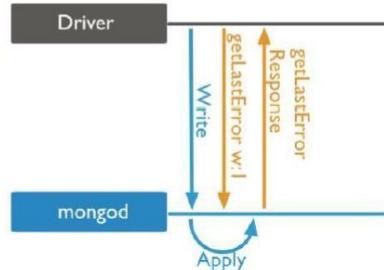


Driver
 mongod
 Write
 Apply
 Unacknowledged
 - Client does not get feedback of writing
 - Very fast writing
 - Possible data loss
 - Networking error
 - Key collusion

1.7 mongodb.pdf page 18 - Week 6 - NoSQL, MongoDB, Document Store

Acknowledged

- Mongod sends receipt
- Client detects errors (networking, key collusion)
- This is the default value
- Writing is not guaranteed!



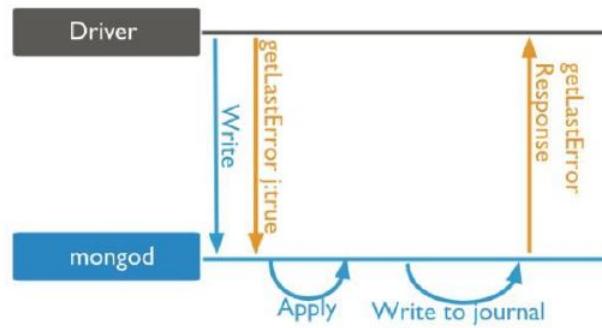
Driver mongod Write Apply getLastError w:1 getLastError Response
 Acknowledged

Mongod sends receipt
 Client detects errors networking , key collusion
 This is the default value
 Writing is not guaranteed!

1.8 mongodb.pdf page 19 - Week 6 - NoSQL, MongoDB, Document Store

Jounalede

- Mongod does not acknowledges, until it does not write into the journal. (logging)
- Journal have to be enabled



Driver mongod Write getLastError j:true Apply Write to journal
getLastError Response

Jounalede

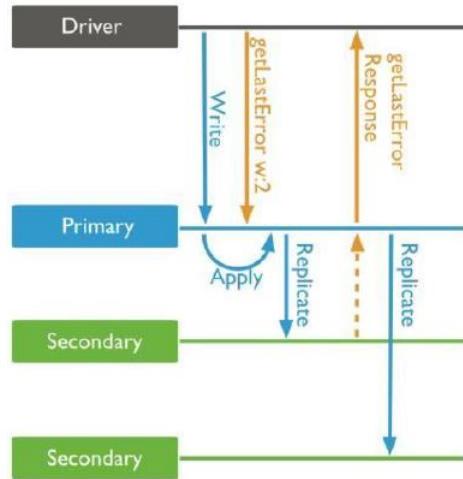
Mongod does not acknowledges , until it does not write into the journal
logging

Journal have to be enabled

1.9 mongodb.pdf page 20 - Week 6 - NoSQL, MongoDB, Document Store

Replica Set Acknowledged

- Can be set, how many replicas have to confirm the writing before it is noted to the client



Driver Primary Secondary
Write Apply getLastError w:2 Replicate getLastError Response
Replica Set Acknowledged
Can be set, how many replicas have to confirm the writing before it is noted to the client

1.10 mongodb.pdf page 22 - Week 6 - NoSQL, MongoDB, Document Store

Extents

my-db.1

my-db.2



- Data and index separate extent
- An extent corresponds to a collection
- A collection can be in multiple extent

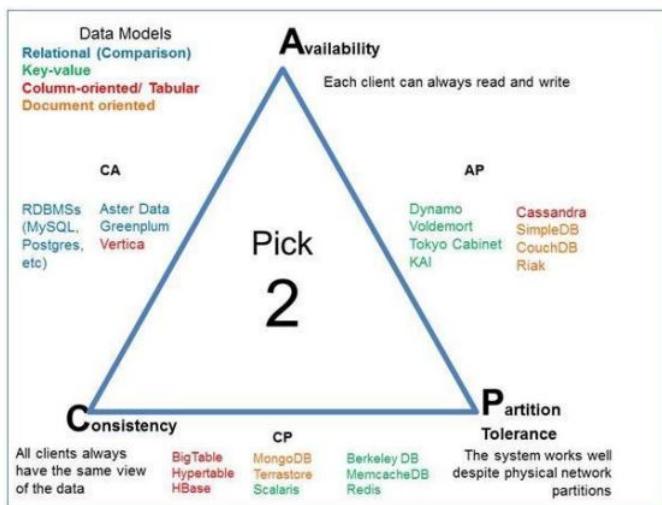
□ Index Extents
■ Data Extents
■ Data Files

Index Extents
 Data Extents
 Data Files
 Extents
 Data and index separate extent
 An extent corresponds to a collection
 A collection can be in multiple extent

2 redis.pdf - Week 8 - NoSQL, Key-value databases, Redis

2.1 redis.pdf page 34 - Week 8 - NoSQL, Key-value databases, Redis

CAP theorem



CAP Theorem
 Data Models Relational (Comparison) Key-value Column-oriented Tabular Document oriented CA CP AP Pick Availability Consistency Partition Tolerance Each client can always read and write RDBMSs MySQL Postgres Aster Data Greenplum Vertica Dynamo Voldemort Tokyo Cabinet KAI Cassandra SampleDB CouchDB Riak All client always have the same view of the data Big Table Hypertable HBase MongoDB Terrastore Scalaris Berkeley DB MemcacheDB Redis The system works well despite physical network partitions

3 Distributed databases.pdf - Week 12 - Data models and Databases Distributed databases

3.1 Distributed databases.pdf page 13 - Week 12 - Data models and Databases Distributed databases

DBMS Implementation Alternatives

