Database Management Systems
CSEP 544

Lecture 7:
Parallel Data processing
Conceptual Design
Class overview

• Data models
  – Relational: SQL, RA, and Datalog
  – NoSQL: SQL++
• RDMBS internals
  – Query processing and optimization
  – Physical design
• Parallel query processing
  – Spark and Hadoop
• Conceptual design
  – E/R diagrams
  – Schema normalization
• Transactions
  – Locking and schedules
  – Writing DB applications
Parallel Data Processing @ 1990
Data: $R(A, B), S(C, D)$
Query: $R(A, B) \bowtie_{B=C} S(C, D)$

Broadcast Join

Why would you want to do this?
Parallel Execution of RA Operators: Partitioned Hash-Join

- **Data**: R(K1, A, B), S(K2, B, C)
- **Query**: R(K1, A, B) \( \bowtie \) S(K2, B, C)
  - Initially, both R and S are partitioned on K1 and K2

- Reshuffle R on R.B and S on S.B
- Each server computes the join locally

\[ B = 10 \quad \text{and} \quad B = 42 \]
HyperCube Join

• Have P number of servers (say P=27 or P=1000)
• How do we compute this Datalog query in one step? 
  \[ Q(x,y,z) = R(x,y), S(y,z), T(z,x) \]

• Organize the P servers into a cube with side \( P^{\frac{1}{3}} \)
  – Thus, each server is uniquely identified by \((i,j,k)\), \(i,j,k \leq P^{\frac{1}{3}}\)

• Step 1:
  – Each server sends \( R(x,y) \) to all servers \((h(x),h(y),*)\)
  – Each server sends \( S(y,z) \) to all servers \((*,h(y),h(z))\)
  – Each server sends \( T(x,z) \) to all servers \((h(x),*,h(z))\)

• Final output:
  – Each server \((i,j,k)\) computes the query \( R(x,y), S(y,z), T(z,x) \) locally

• Analysis: each tuple \( R(x,y) \) is replicated at most \( P^{\frac{1}{3}} \) times
Parallel Data Processing @ 2000
Example

• Counting the number of occurrences of each word in a large collection of documents

• Each Document
  – The key = document id (did)
  – The value = set of words (word)

```java
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += parseInt(v);
  Emit(AsString(result));
```
Interesting Implementation Details

Worker failure:

• Master pings workers periodically,

• If down then reassigns the task to another worker
Interesting Implementation Details

Backup tasks:

• **Straggler** = a machine that takes unusually long time to complete one of the last tasks. E.g.:
  – Bad disk forces frequent correctable errors (30MB/s → 1MB/s)
  – The cluster scheduler has scheduled other tasks on that machine

• Stragglers are a main reason for slowdown

• Solution: *pre-emptive backup execution of the last few remaining in-progress tasks*
Straggler Example

Worker 1

Worker 2

Worker 3

Backup execution

Straggler

Killed

Killed

time
Using MapReduce in Practice:
Implementing RA Operators in MR
Relational Operators in MapReduce

Given relations $R(A,B)$ and $S(B, C)$ compute:

- **Selection**: $\sigma_{A=123}(R)$

- **Group-by**: $\gamma_{A,\text{sum}(B)}(R)$

- **Join**: $R \bowtie S$
Selection $\sigma_{A=123}(R)$

**map**(String value):
  if value.A = 123:
    EmitIntermediate(value.key, value);

**reduce**(String k, Iterator values):
  for each v in values:
    Emit(v);
Selection $\sigma_{A=123}(R)$

map(String value):
if value.A = 123:
    EmitIntermediate(value.key, value);

reduce(String k, Iterator values):
for each v in values:
    Emit(v);

No need for reduce. But need system hacking in Hadoop to remove reduce from MapReduce.
Group By $\gamma_{A, \text{sum}(B)}(R)$

map(String value):
  EmitIntermediate(value.A, value.B);

reduce(String k, Iterator values):
  s = 0
  for each v in values:
    s = s + v
  Emit(k, s);
Join

Two simple parallel join algorithms:

• Partitioned hash-join (we saw it, will recap)

• Broadcast join
\( R(A, B) \bowtie_{B=C} S(C, D) \)

**Partitioned Hash-Join**

Initially, both \( R \) and \( S \) are horizontally partitioned

- Reshuffle \( R \) on \( R.B \) and \( S \) on \( S.B \)
- Each server computes the join locally
\[ \text{R(A,B)} \bowtie_{B=C} \text{S(C,D)} \]

**Partitioned Hash-Join**

**map(String value):**
- case value.relationName of
  - ‘R’: EmitIntermediate(value.B, ('R', value));
  - ‘S’: EmitIntermediate(value.C, ('S', value));

**reduce(String k, Iterator values):**
- R = empty; S = empty;
- for each v in values:
  - case v.type of:
    - ‘R’: R.insert(v)
    - ‘S’: S.insert(v);
- for v1 in R, for v2 in S
  - Emit(v1,v2);
\( R(A,B) \bowtie_{B=C} S(C,D) \)

**Broadcast Join**

- Reshuffle \( R \) on \( R.B \)
- Broadcast \( S \)

Diagram:
- \( R_1, S \)
- \( R_2, S \)
- \( \ldots \)
- \( R_p, S \)
- \( S \)
Broadcast Join

\[
R(A,B) \bowtie_{B=C} S(C,D)
\]

**map(String value):**
- open(S); /* over the network */
- hashTbl = new()
- for each w in S:
  - hashTbl.insert(w.B, w)
- close(S);

- for each v in value:
  - for each w in hashTbl.find(v.B)
  - Emit(v, w);

**reduce(...):**
- /* empty: map-side only */

- map should read several records of R: value = some group of records
- Read entire table S, build a Hash Table
- Hash Join
- R tuples

\[ R(A,B) \bowtie_{B=C} S(C,D) \]
HW6

- HW6 will ask you to write SQL queries and MapReduce tasks using Spark

- You will get to “implement” SQL using MapReduce tasks
  - Can you beat Spark’s implementation?
Conclusions

• MapReduce offers a simple abstraction, and handles distribution + fault tolerance
• Speedup/scaleup achieved by allocating dynamically map tasks and reduce tasks to available server. However, skew is possible (e.g., one huge reduce task)
• Writing intermediate results to disk is necessary for fault tolerance, but very slow.
• Spark replaces this with “Resilient Distributed Datasets” = main memory + lineage
Spark
A Case Study of the MapReduce Programming Paradigm
Parallel Data Processing @ 2010
Issues with MapReduce

• Difficult to write more complex queries

• Need multiple MapReduce jobs: dramatically slows down because it writes all results to disk
Spark

- Open source system from UC Berkeley
- Distributed processing over HDFS
- Differences from MapReduce:
  - Multiple steps, including iterations
  - Stores intermediate results in main memory
  - Closer to relational algebra (familiar to you)
- Details:
  [http://spark.apache.org/examples.html](http://spark.apache.org/examples.html)
Spark

• Spark supports interfaces in Java, Scala, and Python
  – Scala: extension of Java with functions/closures

• We will illustrate use the Spark Java interface in this class

• Spark also supports a SQL interface (SparkSQL), and compiles SQL to its native Java interface
Resilient Distributed Datasets

• RDD = Resilient Distributed Datasets
  – A distributed, immutable relation, together with its lineage
  – Lineage = expression that says how that relation was computed = a relational algebra plan

• Spark stores intermediate results as RDD

• If a server crashes, its RDD in main memory is lost. However, the driver (=master node) knows the lineage, and will simply recompute the lost partition of the RDD
Programming in Spark

• A Spark program consists of:
  – Transformations (map, reduce, join…). Lazy
  – Actions (count, reduce, save…). Eager

• Eager: operators are executed immediately

• Lazy: operators are not executed immediately
  – A operator tree is constructed in memory instead
  – Similar to a relational algebra tree

• What are the benefits of lazy execution?
The RDD Interface
Programming in Spark

- **RDD<T>** = an RDD collection of type T
  - Partitioned, recoverable (through lineage), not nested

- **Seq<T>** = a sequence
  - Local to a server, may be nested
Example

Given a large log file `hdfs://logfile.log` retrieve all lines that:

- Start with “ERROR”
- Contain the string “sqlite”

```scala
s = SparkSession.builder().getOrCreate();

lines = s.read().textFile("hdfs://logfile.log");

errors = lines.filter(l -> l.startsWith("ERROR");

sqlerrors = errors.filter(l -> l.contains("sqlite");

sqlerrors.collect();
```
Example

Given a large log file hdfs://logfile.log retrieve all lines that:

• Start with “ERROR”
• Contain the string “sqlite”

```java
s = SparkSession.builder().getOrCreate();

lines = s.read().textFile("hdfs://logfile.log");

errors = lines.filter(l -> l.startsWith("ERROR"));

sqlerrors = errors.filter(l -> l.contains("sqlite"));

sqlerrors.collect();
```

lines, errors, sqlerrors have type JavaRDD<String>
Example

Given a large log file hdfs://logfile.log retrieve all lines that:

• Start with “ERROR”
• Contain the string “sqlite”

```python
s = SparkSession.builder.getOrCreate();
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l => l.startsWith("ERROR"));
sqlerrors = errors.filter(l => l.contains("sqlite"));
sqlerrors.collect();
```
Example

Recall: anonymous functions (lambda expressions) starting in Java 8

```
errors = lines.filter(l -> l.startsWith("ERROR"));
```

is the same as:

```
class FilterFn implements Function<Row, Boolean>{
  Boolean call (Row r)
  { return r.startsWith("ERROR"); }
}

errors = lines.filter(new FilterFn());
```
Example

Given a large log file hdfs://logfile.log retrieve all lines that:

- Start with “ERROR”
- Contain the string “sqlite”

```
s = SparkSession.builder().getOrCreate();
sqlerrors = s.read().textFile("hdfs://logfile.log")
    .filter(l -> l.startsWith("ERROR"))
    .filter(l -> l.contains("sqlite"))
    .collect();
```

“Call chaining” style
MapReduce Again…

Steps in Spark resemble MapReduce:

- `col.filter(p)` applies in parallel the predicate `p` to all elements `x` of the partitioned collection, and returns a collection with those `x` where `p(x) = true`

- `col.map(f)` applies in parallel the function `f` to all elements `x` of the partitioned collection, and returns a new partitioned collection
Persistence

```scala
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect();
```

If any server fails before the end, then Spark must restart
Persistence

If any server fails before the end, then Spark must restart

```scala
lines = s.read().textFile("hdfs://logfile.log");
ersors = lines.filter(l \rightarrow l.startsWith("ERROR"));
sqllerrors = errors.filter(l \rightarrow l.contains("sqlite"));
sqllerrors.collect();
```
Persistence

If any server fails before the end, then Spark must restart

```scala
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect();
```

Spark can recompute the result from errors
Persistence

If any server fails before the end, then Spark must restart

Spark can recompute the result from errors
Example

R(A,B)  S(A,C)

SELECT count(*) FROM R, S
WHERE R.B > 200 and S.C < 100  and R.A = S.A

R = s.read().textFile("R.csv").map(parseRecord).persist();
S = s.read().textFile("S.csv").map(parseRecord).persist();

Parses each line into an object
persisting on disk
Example

```
SELECT count(*) FROM R, S
WHERE R.B > 200 and S.C < 100 and R.A = S.A

R = s.read().textFile("R.csv").map(parseRecord).persist();
S = s.read().textFile("S.csv").map(parseRecord).persist();
RB = R.filter(t -> t.b > 200).persist();
SC = S.filter(t -> t.c < 100).persist();
J = RB.join(SC).persist();
J.count();
```

**Diagram:**
- **R** (A,B)
  - filter([(a,b)->b>200])
  - **RB**
- **S** (A,C)
  - filter([(b,c)->c<100])
  - **SC**
- **J**
  - join
  - t.split(",") [4]
Recap: Programming in Spark

• A Spark/Scala program consists of:
  – Transformations (map, reduce, join...). **Lazy**
  – Actions (count, reduce, save...). **Eager**

• RDD<T> = an RDD collection of type T
  – Partitioned, recoverable (through lineage), not nested

• Seq<T> = a sequence
  – Local to a server, may be nested
### Transformations:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>map(f : T -&gt; U)</td>
<td>RDD&lt;T&gt; -&gt; RDD&lt;U&gt;</td>
</tr>
<tr>
<td>flatMap(f: T -&gt; Seq(U))</td>
<td>RDD&lt;T&gt; -&gt; RDD&lt;U&gt;</td>
</tr>
<tr>
<td>filter(f:T-&gt;Bool)</td>
<td>RDD&lt;T&gt; -&gt; RDD&lt;T&gt;</td>
</tr>
<tr>
<td>groupByKey()</td>
<td>RDD&lt;(K,V)&gt; -&gt; RDD&lt;(K,Seq[V])&gt;</td>
</tr>
<tr>
<td>reduceByKey(F:(V,V)-&gt; V)</td>
<td>RDD&lt;(K,V)&gt; -&gt; RDD&lt;(K,V)&gt;</td>
</tr>
<tr>
<td>union()</td>
<td>(RDD&lt;T&gt;,RDD&lt;T&gt;) -&gt; RDD&lt;T&gt;</td>
</tr>
<tr>
<td>join()</td>
<td>(RDD&lt;(K,V)&gt;,RDD&lt;(K,W)&gt;) -&gt; RDD&lt;(K,(V,W))&gt;</td>
</tr>
<tr>
<td>cogroup()</td>
<td>(RDD&lt;(K,V)&gt;,RDD&lt;(K,W)&gt;) -&gt; RDD&lt;(K,(Seq&lt;V&gt;,Seq&lt;W&gt;))&gt;</td>
</tr>
<tr>
<td>crossProduct()</td>
<td>(RDD&lt;T&gt;,RDD&lt;U&gt;) -&gt; RDD&lt;(T,U)&gt;</td>
</tr>
</tbody>
</table>

### Actions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>RDD&lt;T&gt; -&gt; Long</td>
</tr>
<tr>
<td>collect()</td>
<td>RDD&lt;T&gt; -&gt; Seq&lt;T&gt;</td>
</tr>
<tr>
<td>reduce(f:(T,T)-&gt;T)</td>
<td>RDD&lt;T&gt; -&gt; T</td>
</tr>
<tr>
<td>save(path:String)</td>
<td>Outputs RDD to a storage system e.g., HDFS</td>
</tr>
</tbody>
</table>
Spark 2.0

The DataFrame and Dataset Interfaces
DataFrames

• Like RDD, also an immutable distributed collection of data

• Organized into *named columns* rather than individual objects
  – Just like a relation
  – Elements are untyped objects called Row’s

• Similar API as RDDs with additional methods
  – `people = spark.read().textFile(...);
    ageCol = people.col("age");
    ageCol.plus(10); // creates a new DataFrame`
Datasets

• Similar to DataFrames, except that elements must be typed objects

• E.g.: Dataset<People> rather than Dataset<Row>

• Can detect errors during compilation time

• DataFrames are aliased as Dataset<Row> (as of Spark 2.0)

• You will use both Datasets and RDD APIs in HW6
Datasets API: Sample Methods

• Functional API
  - `agg(Column expr, Column... exprs)`
    Aggregates on the entire Dataset without groups.
  - `groupBy(String col1, String... cols)`
    Groups the Dataset using the specified columns, so that we can run aggregation on them.
  - `join(Dataset<?, ?> right)`
    Join with another DataFrame.
  - `orderBy(Column... sortExprs)`
    Returns a new Dataset sorted by the given expressions.
  - `select(Column... cols)`
    Selects a set of column based expressions.

• “SQL” API
  - `SparkSession.sql("select * from R");`

• Look familiar?
An Example Application
PageRank

• Page Rank is an algorithm that assigns to each page a score such that pages have higher scores if more pages with high scores link to them

• Page Rank was introduced by Google, and, essentially, defined Google
PageRank toy example

Superstep 0

Superstep 1

Superstep 2

Input graph

http://www.slideshare.net/sscdotopen/large-scale/20
for i = 1 to n:
    r[i] = 1/n

repeat
    for j = 1 to n: contribs[j] = 0
    for i = 1 to n:
        k = links[i].length()
        for j in links[i]:
            contribs[j] += r[i] / k
    for i = 1 to n: r[i] = contribs[i]
until convergence
/* usually 10-20 iterations */

Random walk interpretation:

Start at a random node i
At each step, randomly choose an outgoing link and follow it.

Repeat for a very long time

r[i] = prob. that we are at node i
PageRank

for i = 1 to n:
    r[i] = 1/n

repeat
    for j = 1 to n: contribs[j] = 0
    for i = 1 to n:
        k = links[i].length()
        for j in links[i]:
            contribs[j] += r[i] / k
    for i = 1 to n: r[i] = contribs[i]
until convergence
/* usually 10-20 iterations */

Random walk interpretation:

Start at a random node i
At each step, randomly choose an outgoing link and follow it.

Improvement: with small prob. a restart at a random node.

r[i] = a/N + (1-a)*contribs[i]

where a ∈ (0,1) is the restart probability
```plaintext
for i = 1 to n: 
r[i] = 1/n

repeat
  for j = 1 to n: contribs[j] = 0
  for i = 1 to n:
    k = links[i].length()
    for j in links[i]:
      contribs[j] += r[i] / k
  for i = 1 to n: r[i] = a/N + (1-a)*contribs[i]
  until convergence
/* usually 10-20 iterations */
```
for i = 1 to n:
    r[i] = 1/n

repeat
    for j = 1 to n: contribs[j] = 0
    for i = 1 to n:
        k = links[i].length()
        for j in links[i]:
            contribs[j] += r[i] / k
    for i = 1 to n: r[i] = a/N + (1-a)*contribs[i]
until convergence
/* usually 10-20 iterations */

// spark
links = spark.read().textFile(..).map(...);
ranks = // RDD of (URL, 1/n) pairs
for (k = 1 to ITERATIONS) {
    // Build RDD of (targetURL, float) pairs
    // with contributions sent by each page
    contribs = links.join(ranks).flatMap {
        (url, lr) -> // lr: a (link, rank) pair
            links.map(dest ->
                (dest, lr._2/outlinks.size())
            )
    }
    // Sum contributions by URL and get new ranks
    ranks = contribs.reduceByKey((x,y) -> x+y)
        .mapValues(sum -> a/N + (1-a)*sum)
}
for i = 1 to n:
   r[i] = 1/n
repeat
   for j = 1 to n: contribs[j] = 0
   for i = 1 to n:
      k = links[i].length()
      for j in links[i]:
         contribs[j] += r[i] / k
   for i = 1 to n: r[i] = a/N + (1-a)*contribs[i]
until convergence
/* usually 10-20 iterations */
Conclusions

• Parallel databases
  – Predefined relational operators
  – Optimization
  – Transactions

• MapReduce
  – User-defined map and reduce functions
  – Must implement/optimize manually relational ops
  – No updates/transactions

• Spark
  – Predefined relational operators
  – Must optimize manually
  – No updates/transactions
Conceptual Design
Class overview

• Data models
  – Relational: SQL, RA, and Datalog
  – NoSQL: SQL++

• RDBMS internals
  – Query processing and optimization
  – Physical design

• Parallel query processing
  – Spark and Hadoop

• Conceptual design
  – E/R diagrams
  – Schema normalization

• Transactions
  – Locking and schedules
  – Writing DB applications
Database Design

What it is:

• Starting from scratch, design the database schema: relation, attributes, keys, foreign keys, constraints etc

Why it’s hard

• The database will be in operation for a very long time (years). Updating the schema while in production is very expensive (why?)
Database Design

• Consider issues such as:
  – What entities to model
  – How entities are related
  – What constraints exist in the domain

• Several formalisms exists
  – We discuss E/R diagrams
  – UML, model-driven architecture

• Reading: Sec. 4.1-4.6
Database Design Process

Conceptual Model:

- Relational Model: Tables + constraints
- And also functional dep.

Normalization:
- Eliminates anomalies

Conceptual Schema

Physical storage details

Physical Schema
Entity / Relationship Diagrams

- Entity set = a class
  - An entity = an object

- Attribute

- Relationship
Keys in E/R Diagrams

- Every entity set must have a key
What is a Relation?

- A mathematical definition:
  - if A, B are sets, then a relation R is a subset of A X B

- \( A = \{1, 2, 3\}, \quad B = \{a, b, c, d\}, \)
  - \( A \times B = \{(1, a), (1, b), \ldots, (3, d)\} \)
  - \( R = \{(1, a), (1, c), (3, b)\} \)

- **makes** is a subset of **Product X Company**:

![Diagram of a relation between Product and Company]
Multiplicities of E/R Relations

- **one-one:**
  - 1
  - 2
  - 3
  - a
  - b
  - c
  - d

- **many-one**
  - 1
  - 2
  - 3
  - a
  - b
  - c
  - d

- **many-many**
  - 1
  - 2
  - 3
  - a
  - b
  - c
  - d

*CSEP 544 - Fall 2017*
What does this say?
Multi-way Relationships

How do we model a purchase relationship between buyers, products and stores?

Can still model as a mathematical set (How?)

As a set of triples \( \subseteq \text{Person} \times \text{Product} \times \text{Store} \)
Q: What does the arrow mean?

A: A given person buys a given product from at most one store.

[Fine print: Arrow pointing to E means that if we select one entity from each of the other entity sets in the relationship, those entities are related to at most one entity in E]
Q: What does the arrow mean?

A: A given person buys a given product from at most one store AND every store sells to every person at most one product.
Converting Multi-way Relationships to Binary

Arrows go in which direction?
Converting Multi-way Relationships to Binary

Make sure you understand why!
3. Design Principles

What’s wrong?

Product - Purchase - Person

Country - President - Person

Moral: Be faithful to the specifications of the application!
Design Principles:
What’s Wrong?

Moral: pick the right kind of entities.
Design Principles: What’s Wrong?

Moral: don’t complicate life more than it already is.
From E/R Diagrams to Relational Schema

- Entity set $\rightarrow$ relation
- Relationship $\rightarrow$ relation
Entity Set to Relation

Product(prod-ID, category, price)

<table>
<thead>
<tr>
<th>prod-ID</th>
<th>category</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo55</td>
<td>Camera</td>
<td>99.99</td>
</tr>
<tr>
<td>Pokemn19</td>
<td>Toy</td>
<td>29.99</td>
</tr>
</tbody>
</table>
N-N Relationships to Relations

Represent this in relations
Orders (prod-ID, cust-ID, date)
Shipment (prod-ID, cust-ID, name, date)
Shipping-Co (name, address)

<table>
<thead>
<tr>
<th>prod-ID</th>
<th>cust-ID</th>
<th>name</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo55</td>
<td>Joe12</td>
<td>UPS</td>
<td>4/10/2011</td>
</tr>
<tr>
<td>Gizmo55</td>
<td>Joe12</td>
<td>FEDEX</td>
<td>4/9/2011</td>
</tr>
</tbody>
</table>
N-1 Relationships to Relations

Represent this in relations
Orders\(\langle\text{prod-ID, cust-ID, date1, name, date2}\rangle\)

Shipping-Co\(\langle\text{name, address}\rangle\)

Remember: no separate relations for many-one relationship
Multi-way Relationships to Relations

Product
- prod-ID
- price

Purchase

Person
- ssn
- name

Store
- name
- address

Try this at home!

Purchase(prod-ID, ssn, name)
Modeling Subclasses

Some objects in a class may be special
• define a new class
• better: define a subclass

Products

Software products

Educational products

So --- we define subclasses in E/R
Subclasses to Relations

<table>
<thead>
<tr>
<th>Product</th>
<th>Name</th>
<th>Price</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>99</td>
<td>gadget</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td>49</td>
<td>photo</td>
<td></td>
</tr>
<tr>
<td>Toy</td>
<td>39</td>
<td>gadget</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software Product</th>
<th>Name</th>
<th>platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td></td>
<td>unix</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Educational Product</th>
<th>Name</th>
<th>Age Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td></td>
<td>toddler</td>
</tr>
<tr>
<td>Toy</td>
<td></td>
<td>retired</td>
</tr>
</tbody>
</table>

Other ways to convert are possible

CSEP 544 - Fall 2017
Modeling Union Types with Subclasses

Say: each piece of furniture is owned either by a person or by a company.
Modeling Union Types with Subclasses

Say: each piece of furniture is owned either by a person or by a company

Solution 1. Acceptable but imperfect (What’s wrong?)
Modeling Union Types with Subclasses

Solution 2: better, more laborious
Weak Entity Sets

Entity sets are weak when their key comes from other classes to which they are related.

Team(sport, number, universityName)
University(name)