Normalization

Outline

Mechanisms and definitions to work with FDs
  » Closures, candidate keys, canonical covers etc...
  » Armstrong axioms

Decompositions
  » Loss-less decompositions, Dependency-preserving decompositions

BCNF
  » How to achieve a BCNF schema

BCNF may not preserve dependencies

3NF: Solves the above problem

BCNF allows for redundancy

4NF: Solves the above problem
### 3NF

**Definition:** *Prime attributes*

An attribute that is contained in a candidate key for R

**Example 1:**
- \( R = \{A, B, C, D, E, H\} \), \( F = \{A \rightarrow BC, E \rightarrow HA\} \),
- Candidate keys = \{ED\}
- Prime attributes: D, E

**Example 2:**
- \( R = \{J, K, L\} \), \( F = \{JK \rightarrow L, L \rightarrow K\} \),
- Candidate keys = \{JL, JK\}
- Prime attributes: J, K, L

**Observation/Intuition:**

1. A *key* has no redundancy (is not repeated in a relation)
2. A *prime attribute* has limited redundancy

---

### 3NF

\( R \) is in *3NF (3rd Normal Form)* if:

Given a relation schema \( R \), and a set of functional dependencies \( F \), if every FD, \( \alpha \rightarrow \beta \), is either:

- Trivial, or
- \( \alpha \) is a *superkey* of \( R \), or
- *All attributes in* \((\beta - \alpha)\) *are prime*

**Why is 3NF good?**

- Preserves dependencies.
3NF and Redundancy

Why does redundancy arise?
» Given a FD, $\alpha \rightarrow \beta$, if $\alpha$ is repeated ($\beta - \alpha$) has to be repeated
  • If rule 1 is satisfied, ($\beta - \alpha$) is empty, so not a problem.
  • If rule 2 is satisfied, $\alpha$ can't be repeated, so this doesn't happen either
  • If not, rule 3 says ($\beta - \alpha$) must contain only prime attributes
» This limits the redundancy somewhat.

3NF relaxes BCNF by allowing some (hopefully limited) redundancy

Why good?
» There always exists a dependency-preserving lossless decomposition in 3NF.

Decomposing into 3NF

let $F_c$ be a canonical cover for $F$;
i := 0;
for each functional dependency $\alpha \rightarrow \beta$ in $F_c$
  i := i + 1;
  $R_i := \alpha \beta$;
if none of the schemas $R_j$, $j = 1, 2, \ldots, i$ contains a candidate key for $R$
  then
    i := i + 1;
    $R_i :=$ any candidate key for $R$;
  /* Optionally, remove redundant relations */
repeat
  if any schema $R_j$ is contained in another schema $R_k$
    then
      /* Delete $R_j$ */
      $R_j := R_i$;
      i := i - 1;
until no more $R_j$s can be deleted
return ($R_1, R_2, \ldots, R_i$)

Figure 8.12 Dependency-preserving, lossless decomposition into 3NF.
3NF Example

(R) = (A, B, C, D, E, F, G, H)

Function Dependencies

F = {A→CGH, AD→C, DE→F, G→H}

- R1 = {ACGH}, R2 = {ADC}, R3 = {DEF}, R4 = {GH}
- R1 = {ACGH}, R2 = {ADC}, R3 = {DEF}, R5 = {GH}, R6 = {ABDE}
- R1 = {ACGH}, R2 = {ADC}, R3 = {DEF}, R5 = {ABDE}

> Somewhat better if start from canonical cover: F' =

- {A→CGH, AD→C, DE→F, G→H}
- {A→CGH, AD→C, DE→F, G→H}
- {A→CG, DE→F, G→H}
  - H is extra in A→CGH, D extra in AD→C - then merge w/ A→CG
- R1 = {ACG}, R2 = {DEF}, R3 = {GH}
- R1 = {ACG}, R2 = {DEF}, R3 = {GH}, R4 = {ABDE}

> Lossless: Each has a single FD that is a key
> Preserves dependencies: each carried through a single subrelation

Outline

Mechanisms and definitions to work with FDs
  » Closures, candidate keys, canonical covers etc...
  » Armstrong axioms

Decompositions
  » Loss-less decompositions, Dependency-preserving decompositions

BCNF
  » How to achieve a BCNF schema

BCNF may not preserve dependencies

3NF: Solves the above problem

BCNF allows for redundancy

4NF: Solves the above problem
BCNF and redundancy

<table>
<thead>
<tr>
<th>MovieTitle</th>
<th>MovieYear</th>
<th>StarName</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star wars</td>
<td>1977</td>
<td>Harrison Ford</td>
<td>Address 1, LA</td>
</tr>
<tr>
<td>Star wars</td>
<td>1977</td>
<td>Harrison Ford</td>
<td>Address 2, FL</td>
</tr>
<tr>
<td>Indiana Jones</td>
<td>198x</td>
<td>Harrison Ford</td>
<td>Address 1, LA</td>
</tr>
<tr>
<td>Indiana Jones</td>
<td>198x</td>
<td>Harrison Ford</td>
<td>Address 2, FL</td>
</tr>
<tr>
<td>Witness</td>
<td>19xx</td>
<td>Harrison Ford</td>
<td>Address 1, LA</td>
</tr>
<tr>
<td>Witness</td>
<td>19xx</td>
<td>Harrison Ford</td>
<td>Address 2, FL</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Lot of redundancy

FDs ? No non-trivial FDs.

So the schema is trivially in BCNF (and 3NF)

What went wrong ?

Multi-valued Dependencies

The redundancy is because of multi-valued dependencies

Denoted:

» starname ➔ address
» starname ➔ movietitle, movieyear

Does not happen if schema is constructed from an E/R diagram

Functional dependencies are a special case of multi-valued dependencies
Outline

Mechanisms and definitions to work with FDs
  » Closures, candidate keys, canonical covers etc...
  » Armstrong axioms

Decompositions
  » Loss-less decompositions, Dependency-preserving decompositions

BCNF
  » How to achieve a BCNF schema

BCNF may not preserve dependencies

3NF: Solves the above problem

BCNF allows for redundancy

4NF: Solves the above problem

4NF

Similar to BCNF, except with MVDs instead of FDs.

Given a relation schema $R$, and a set of multi-valued dependencies $F$, if every MVD, $A \rightarrow\rightarrow B$, is either:
  1. Trivial, or
  2. $A$ is a superkey of $R$
  3. Then, $R$ is in 4NF (4th Normal Form)

4NF $\Rightarrow$ BCNF $\Rightarrow$ 3NF $\Rightarrow$ 2NF $\Rightarrow$ 1NF:
  » If a schema is in 4NF, it is in BCNF.
  » If a schema is in BCNF, it is in 3NF.

Other way round is not necessarily true.
Comparing the normal forms

<table>
<thead>
<tr>
<th></th>
<th>3NF</th>
<th>BCNF</th>
<th>4NF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eliminates redundancy because of FD's</td>
<td>Mostly</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Eliminates redundancy because of MVD’s</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Preserves FDs</td>
<td>Yes.</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Preserves MVDs</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
</tbody>
</table>

4NF is typically desired and achieved.

A good E/R diagram won’t generate non-4NF relations at all

Choice between 3NF and BCNF is up to the designer

Database design process

So three ways to come up with a schema:

1. **Using E/R diagram**
   - If good, then little normalization is needed
   - Tends to generate 4NF designs

2. **A universal relation** $R$ that contains all attributes.
   - Called universal relation approach
   - Note that MVDs will be needed in this case

3. **An ad hoc scheme** that is then normalized
   - MVDs may be needed in this case
Recap

What about 1\textsuperscript{st} and 2\textsuperscript{nd} normal forms?

1NF:
» Essentially says that no set-valued attributes allowed
» Formally, a domain is called \textit{atomic} if the elements of the domain are considered indivisible
» A schema is in 1NF if the domains of all attributes are atomic
» We assumed 1NF throughout the discussion
  • Non 1NF is just not a good idea

2NF:
» Mainly historic interest
» See Exercise 7.15 in the book

Recap

We would like our relation schemas to:
» \textit{Not allow potential redundancy} because of FDs or MVDs
» Be dependency-preserving:
  • Make it easy to check for dependencies
  • Since they are a form of integrity constraints

Functional Dependencies/Multi-valued Dependencies
» Domain knowledge about the data properties

Normal forms
» Defines the rules that schemas must follow
» 4NF is preferred, but 3NF is sometimes used instead
Recap

Denormalization
» After doing the normalization, we may have too many tables
» We may *denormalize* for performance reasons
  • Too many tables → too many joins during queries
» A better option is to use *views* instead
  • If a specific set of tables is joined often, create a view on the join

More advanced normal forms
» project-join normal form (PJNF or 5NF)
» domain-key normal form
» Rarely used in practice

Big Data Processing with MapReduce and Spark
The Big Data Problem

Data is growing faster than computation speeds

Growing data sources
  » Web, mobile, scientific, ...

Cheap storage
  » Doubling every 18 months

Stalling CPU speeds
  » Even multicores not enough
Examples

Facebook’s daily logs: 60 TB
1000 genomes project: 200 TB
Google web index: 10+ PB

Cost of 1 TB of disk: $50
Time to read 1 TB from disk: 6 hours (50 MB/s)

The Big Data Problem

Single machine can no longer process or even store all the data!

Only solution is to distribute over large clusters
Google Datacenter

How do we program this thing?

Traditional Network Programming

Message-passing between nodes

Really hard to do at scale:
» How to split problem across nodes?
  • Must consider network, data locality
» How to deal with failures?
  • 1 server fails every 3 years => 10K nodes see 10 faults/day
» Even worse: stragglers (node is not failed, but slow)

Almost nobody does message passing!
Data-Parallel Models

Restrict the programming interface so that the system can do more automatically

“Here’s an operation, run it on all of the data”
» I don’t care where it runs (you schedule that)
» In fact, feel free to run it twice on different nodes

Biggest example: MapReduce
» (Hadoop is open source copy)

MapReduce

First widely popular programming model for data-intensive apps on clusters

Published by Google in 2004
» Processes 20 PB of data / day

Popularized by open-source Hadoop project
» 40,000 nodes at Yahoo!, 70 PB at Facebook
Outline

The big data problem

MapReduce model

Limitations of MapReduce

Spark model

MapReduce Programming Model

Data type: key-value records

Map function:

\[(K_{in}, V_{in}) \mapsto \text{list}(K_{inter}, V_{inter})\]

Reduce function:

\[(K_{inter}, \text{list}(V_{inter})) \mapsto \text{list}(K_{out}, V_{out})\]
Example: Word Count

```python
def mapper(line):
    for word in line.split():
        output(word, 1)

def reducer(key, values):
    output(key, sum(values))
```

Word Count Execution

- **Input**: the quick brown fox, the fox ate the mouse, how now brown cow
- **Map**: the, 1 brown, 1 fox, 1
- **Shuffle & Sort**: the, 1 brown, 1 fox, 1
- **Reduce**: brown, 2 fox, 2 how, 1 now, 1 the, 3
- **Output**: brown, 2 fox, 2 how, 1 now, 1 the, 3
MapReduce Execution

Automatically split work into many small *tasks*
Send map tasks to nodes based on data locality
Load-balance dynamically as tasks finish

Fault Recovery

1. If a task crashes:
   » Retry on another node
     • OK for a map because it had no dependencies
     • OK for reduce because map outputs are on disk
   » If the same task repeatedly fails, end the job

Requires user code to be *deterministic*
Fault Recovery

2. If a node crashes:
   » Relaunch its current tasks on other nodes
   » Relaunch any maps the node previously ran
     • Necessary because their output files were lost along with the crashed node

Fault Recovery

3. If a task is going slowly (straggler):
   » Launch second copy of task on another node
   » Take the output of whichever copy finishes first, and kill the other one
Example Applications

1. Search

Input: (lineNumber, line) records

Output: lines matching a given pattern

Map:

\[
\text{if(line matches pattern):} \\
\quad \text{output(line)}
\]

Reduce: identity function

– Alternative: no reducer (map-only job)
2. Sort

**Input:** (key, value) records

**Output:** same records, sorted by key

**Map:** identity function

**Reduce:** identify function

3. Inverted Index

**Input:** (filename, text) records

**Output:** list of files containing each word

**Map:**

```python
foreach word in text.split():
    output(word, filename)
```

**Reduce:**

```python
def reduce(word, filenames):
    output(word, unique(filenames))
```
Inverted Index Example

hamlet.txt
  to be or not to be

12th.txt
  be not afraid of greatness

Summary

By providing a data-parallel model, MapReduce greatly simplified cluster programming:

» Automatic division of job into tasks
» Locality-aware scheduling
» Load balancing
» Recovery from failures & stragglers

But... the story doesn’t end here!
Outline

The big data problem
MapReduce model

Limitations of MapReduce
Spark model

When an Abstraction is Useful…

People want to compose it!

Most real applications require multiple MR steps
  » Google indexing pipeline: 21 steps
  » Analytics queries (e.g. sessions, top K): 2-5 steps
  » Iterative algorithms (e.g. PageRank): 10’s of steps

Problems: programmability & performance
**Programmability**

Multi-step jobs create spaghetti code
  » 21 MR steps -> 21 mapper and reducer classes

Lots of boilerplate wrapper code per step

API doesn’t provide type safety

**Performance**

MR only provides one pass of computation
  » Must write out data to file system in-between

Expensive for apps that need to *reuse* data
  » Multi-step algorithms (e.g. PageRank)
  » Interactive data mining (many queries on same data)

Users often hand-optimize by merging steps
Outline

The big data problem
MapReduce model
Limitations of MapReduce
Spark model

Spark

Aims to address both problems

Programmability: clean, functional API
  » Parallel transformations on collections
  » 5-10x less code than MR
  » Available in Scala, Java and Python

Performance:
  » In-memory computing primitives
  » Automatic optimization across operators
Spark Programmability

Google MapReduce WordCount:

```cpp
#include "mapreduce/mapreduce.h"

// User's map function
class SplitWords: public Mapper {
public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while (i < n && isspace(text[i])) ++i;
      // Find word end
      int start = i;
      while (i < n && !isspace(text[i])) ++i;
      if (start < i) Emit(text.substr(start, i - start), "1");
    }
  }
REGISTER_MAPPERS(SplitWords);

// User's reduce function
class Sum: public Reducer {
public:
  virtual void Reduce(ReduceInput* input) {
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }
    // Emit sum for input
    Emit(IntToString(value));
  }
REGISTER_REDUCERS(Sum);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);
  MapReduceSpecification spec;
  for (int i = 1; i < argc; ++i) {
    MapReduceInput* in = spec.add_input();
    in->set_format("text");
    in->set_filepattern(argv[i]);
    in->set_mapper_class("SplitWords");
  }
  // Specify the output files
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Sum");
  // Do partial sums within map
  out->set_combiner_class("Sum");
  // Tuning parameters
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);
  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();
  return 0;
}
```

Spark WordCount:

```scala
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
counts.save("out.txt")
```
Spark Performance

Iterative algorithms:

- **K-means Clustering**
  - Hadoop MR: 4.1 sec
  - Spark: 121 sec

- **Logistic Regression**
  - Hadoop MR: 0.96 sec
  - Spark: 80 sec

Spark Concepts

**Resilient distributed datasets (RDDs)**
- Immutable, partitioned collections of objects
- May be cached in memory for fast reuse

**Operations on RDDs**
- *Transformations* (build RDDs), *actions* (compute results)

**Restricted shared variables**
- Broadcast, accumulators
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
def log_mining(spark):
    lines = spark.textFile("hdfs://...")
    errors = lines.filter(_.startsWith("ERROR"))
    messages = errors.map(_.split(\'\t\')(2))
    messages.cache()

    foo_count = messages.filter(_.contains("foo")).count
    bar_count = messages.filter(_.contains("bar")).count

    print(f"Result: search 1TB data in {foo_count} - {bar_count} sec")
```

Result: search 1TB data in 5-7 sec (vs 170 sec for on-disk data)

Example: Logistic Regression

Goal: find best line separating two sets of points

![Logistic Regression Diagram]
Example: Logistic Regression

```scala
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p => (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x)
  w = gradient.reduce(_ + _)
}

println("Final w: "+ w)
```

Logistic Regression Performance

- First iteration: 80 s
- Further iterations: 1 s
- Average iteration time: 110 s
### Other RDD Operations

<table>
<thead>
<tr>
<th>Transformations</th>
<th>map</th>
<th>flatMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define a new RDD)</td>
<td>filter</td>
<td>union</td>
</tr>
<tr>
<td></td>
<td>sample</td>
<td>join</td>
</tr>
<tr>
<td></td>
<td>groupByKey</td>
<td>cross</td>
</tr>
<tr>
<td></td>
<td>reduceByKey</td>
<td>mapValues</td>
</tr>
<tr>
<td></td>
<td>cogroup</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actions</th>
<th>collect</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>(output a result)</td>
<td>reduce</td>
<td>saveAsTextFile</td>
</tr>
<tr>
<td></td>
<td>take</td>
<td>saveAsHadoopFile</td>
</tr>
<tr>
<td></td>
<td>fold</td>
<td></td>
</tr>
</tbody>
</table>

### Spark in Java and Python

```java
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
    Boolean call(String s) {
        return s.contains("error");
    }
}).count();
```

```python
lines = sc.textFile(...)
lines.filter(lambda x: "error" in x).count()
```
Accumulators

Apart from broadcast, another common sharing pattern is aggregation
  » Add up multiple statistics about data
  » Count various events for debugging

Spark’s reduce operation does aggregation, but accumulators are another nice way to express it

Usage

```scala
val badRecords = sc.accumulator(0)
val badBytes = sc.accumulator(0.0)
records.filter(r => {
  if (isBad(r)) {
    badRecords += 1
    badBytes += r.size
    false
  } else {
    true
  }
}).save(...)

printf("Total bad records: %d, avg size: %f\n",
  badRecords.value, badBytes.value / badRecords.value)
```
Accumulator Rules

Create with SparkContext.accumulator(initialVal)

“Add” to the value with += inside tasks
  » Each task’s effect only counted once

Access with .value, but only on master
  » Exception if you try it on workers

Retains efficiency and fault tolerance!