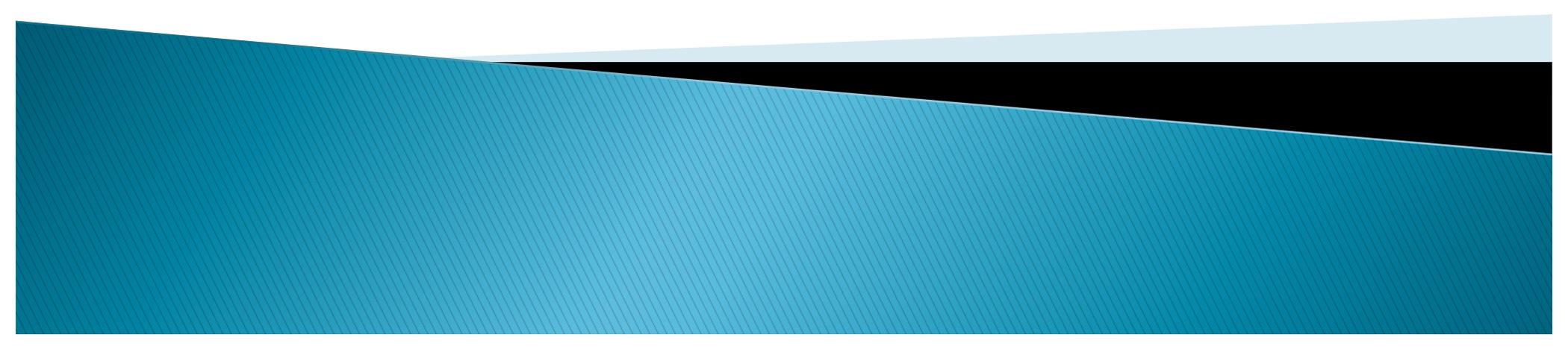


CMSC424: Database Design

Normalization

March 2, 2020

Instructor: Amol Deshpande
amol@cs.umd.edu



Plan for Today

- ▶ **Wrap up Normalization**
 - ▶ **Projects**
 - Will start using ELMS for announcements
 - Regrading etc.
 - Project 3 uploaded – will post officially after a review today
 - ▶ **Midterm 1 on Wednesday: Questions?**
 - ▶ **Next topic:**
 - How to “execute” an SQL Query?
 - **Today: General background and alternatives**
- 

Approach

1. We will encode and list all our knowledge about the schema

- Functional dependencies (FDs)
- Also:
 - Multi-valued dependencies (briefly discuss later)
 - Join dependencies etc...

2. We will define a set of rules that the schema must follow to be considered good

- “Normal forms”: 1NF, 2NF, 3NF, BCNF, 4NF, ...
- A normal form specifies constraints on the schemas and FDs

3. If not in a “normal form”, we modify the schema



BCNF: Boyce-Codd Normal Form

- ▶ A relation schema R is “in BCNF” if:
 - Every functional dependency $A \rightarrow B$ that holds on it is **EITHER**:
 1. Trivial **OR**
 2. A is a *superkey* of R

- ▶ Why is BCNF good ?
 - Guarantees that there can be no redundancy because of a functional dependency
 - Consider a relation $r(A, B, C, D)$ with functional dependency $A \rightarrow B$ and two tuples: $(a1, b1, c1, d1)$, and $(a1, b1, c2, d2)$
 - $b1$ is repeated because of the functional dependency
 - BUT this relation is not in BCNF
 - $A \rightarrow B$ is neither trivial nor is A a superkey for the relation

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**

1. Closure

- ▶ Given a set of functional dependencies, F , its *closure*, F^+ , is all FDs that are implied by FDs in F .
 - e.g. If $A \rightarrow B$, and $B \rightarrow C$, then clearly $A \rightarrow C$
- ▶ We can find F^+ by applying **Armstrong's Axioms**:
 - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
 - if $\alpha \rightarrow \beta$, then $\gamma \alpha \rightarrow \gamma \beta$ **(augmentation)**
 - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**
- ▶ These rules are
 - sound (generate only functional dependencies that actually hold)
 - complete (generate all functional dependencies that hold)

Additional rules

- ▶ If $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$, then $\alpha \rightarrow \beta\gamma$ (**union**)
- ▶ If $\alpha \rightarrow \beta\gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ (**decomposition**)
- ▶ If $\alpha \rightarrow \beta$ and $\gamma\beta \rightarrow \delta$, then $\alpha\gamma \rightarrow \delta$ (**pseudotransitivity**)

- ▶ The above rules can be inferred from Armstrong's axioms.

Example

▶ $R = (A, B, C, G, H, I)$

$F = \{ A \rightarrow B$

$A \rightarrow C$

$CG \rightarrow H$

$CG \rightarrow I$

$B \rightarrow H\}$

▶ Some members of F^+

◦ $A \rightarrow H$

• by transitivity from $A \rightarrow B$ and $B \rightarrow H$

◦ $AG \rightarrow I$

• by augmenting $A \rightarrow C$ with G , to get $AG \rightarrow CG$
and then transitivity with $CG \rightarrow I$

◦ $CG \rightarrow HI$

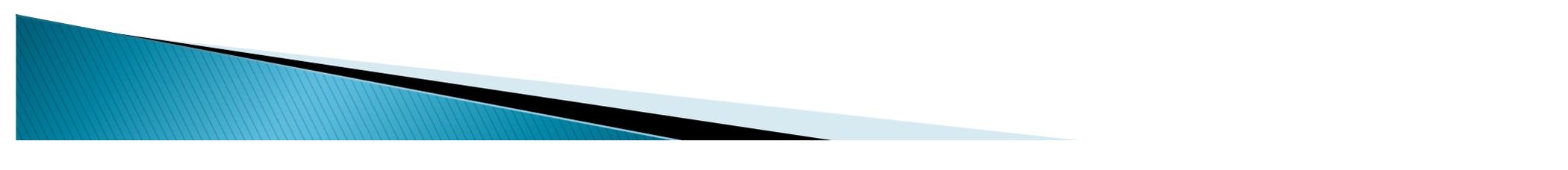
• by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$,
and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$,
and then transitivity

2. Closure of an attribute set

- ▶ Given a set of attributes A and a set of FDs F , *closure of A under F* is the set of all attributes implied by A
- ▶ In other words, the largest B such that: $A \rightarrow B$
- ▶ Redefining *super keys*:
 - *The closure of a super key is the entire relation schema*
- ▶ Redefining *candidate keys*:
 1. It is a super key
 2. No subset of it is a super key

Computing the closure for A

- ▶ Simple algorithm
- ▶ 1. Start with $B = A$.
- ▶ 2. Go over all functional dependencies, $\beta \rightarrow \gamma$, in F^+
- ▶ 3. If $\beta \subseteq B$, then
 Add γ to B
- ▶ 4. Repeat till B changes



Example

▶ $R = (A, B, C, G, H, I)$

$F = \{ A \rightarrow B$

$A \rightarrow C$

$CG \rightarrow H$

$CG \rightarrow I$

$B \rightarrow H\}$

▶ $(AG)^+ ?$

◦ 1. result = AG

◦ 2. result = ABCG ($A \rightarrow C$ and $A \rightarrow B$)

◦ 3. result = ABCGH ($CG \rightarrow H$ and $CG \subseteq AGBC$)

◦ 4. result = ABCGHI ($CG \rightarrow I$ and $CG \subseteq AGBCH$)

▶ Is (AG) a candidate key ?

1. It is a super key.

2. $(A^+) = ABCH$, $(G^+) = G$.

YES.

Uses of attribute set closures

- ▶ Determining *superkeys and candidate keys*
- ▶ Determining if $A \rightarrow B$ is a valid FD
 - Check if A^+ contains B
- ▶ Can be used to compute F^+

3. Extraneous Attributes

- ▶ Consider F , and a functional dependency, $A \rightarrow B$.
- ▶ “Extraneous”: Are there any attributes in A or B that can be safely removed ?

Without changing the constraints implied by F

- ▶ Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
 - C is extraneous in $AB \rightarrow CD$ since $AB \rightarrow C$ can be inferred even after deleting C
 - ie., given: $A \rightarrow C$, and $AB \rightarrow D$, we can use Armstrong Axioms to infer $AB \rightarrow CD$

4. Canonical Cover

- ▶ A *canonical cover* for F is a set of dependencies F_c such that
 - F logically implies all dependencies in F_c , and
 - F_c logically implies all dependencies in F , and
 - No functional dependency in F_c contains an extraneous attribute, and
 - Each left side of functional dependency in F_c is unique
- ▶ In some (vague) sense, it is a *minimal* version of F
- ▶ Read up algorithms to compute F_c

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
- 

Loss-less Decompositions

- ▶ Definition: A decomposition of R into $(R1, R2)$ is called *lossless* if, for all legal instance of $r(R)$:

$$r = \prod_{R1}(r) \bowtie \prod_{R2}(r)$$

- ▶ In other words, projecting on $R1$ and $R2$, and *joining back*, results in the relation you started with
- ▶ Rule: A decomposition of R into $(R1, R2)$ is *lossless*, iff:

$$R1 \cap R2 \rightarrow R1 \quad \text{or} \quad R1 \cap R2 \rightarrow R2$$

in $F+$.



Dependency-preserving Decompositions

Is it easy to check if the dependencies in F hold ?

Okay as long as the dependencies can be checked in the same table.

Consider $R = (A, B, C)$, and $F = \{A \rightarrow B, B \rightarrow C\}$

1. Decompose into $R1 = (A, B)$, and $R2 = (A, C)$

Lossless ? Yes.

But, makes it hard to check for $B \rightarrow C$

The data is in multiple tables.

2. On the other hand, $R1 = (A, B)$, and $R2 = (B, C)$,

is both lossless and dependency-preserving

Really ? What about $A \rightarrow C$?

If we can check $A \rightarrow B$, and $B \rightarrow C$, $A \rightarrow C$ is implied.

Dependency-preserving Decompositions

▶ Definition:

- Consider decomposition of R into R_1, \dots, R_n .
- Let F_i be the set of dependencies F^+ that include only attributes in R_i .

- ▶ The decomposition is **dependency preserving**, if

$$(F_1 \cup F_2 \cup \dots \cup F_n)^+ = F^+$$

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

- ▶ Given a relation schema R , and a set of functional dependencies F , if every FD, $A \rightarrow B$, is either:
 1. Trivial
 2. A is a *superkey* of RThen, R is in **BCNF (Boyce-Codd Normal Form)**

- ▶ What if the schema is not in BCNF ?
 - *Decompose (split) the schema into two pieces.*
 - Careful: you want the decomposition to be lossless



Achieving BCNF Schemas

For all dependencies $A \rightarrow B$ in F^+ , check if A is a superkey

By using attribute closure

If not, then

Choose a dependency in F^+ that breaks the BCNF rules, say $A \rightarrow B$

Create $R_1 = A B$

Create $R_2 = A (R - B - A)$

Note that: $R_1 \cap R_2 = A$ and $A \rightarrow AB (= R_1)$, so this is lossless decomposition

Repeat for R_1 , and R_2

By defining F_1^+ to be all dependencies in F that contain only attributes in R_1

Similarly F_2^+



Example 1

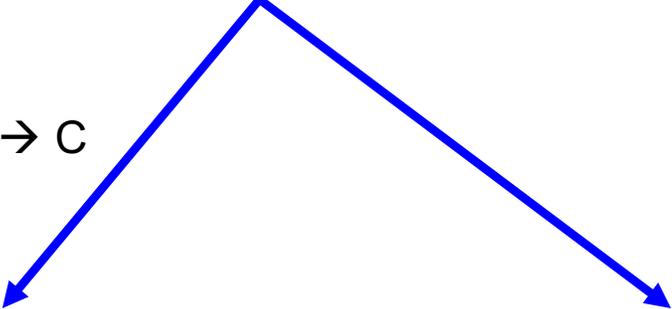
$R = (A, B, C)$

$F = \{A \rightarrow B, B \rightarrow C\}$

Candidate keys = $\{A\}$

BCNF = No. $B \rightarrow C$ violates.

$B \rightarrow C$



$R1 = (B, C)$

$F1 = \{B \rightarrow C\}$

Candidate keys = $\{B\}$

BCNF = true

$R2 = (A, B)$

$F2 = \{A \rightarrow B\}$

Candidate keys = $\{A\}$

BCNF = true

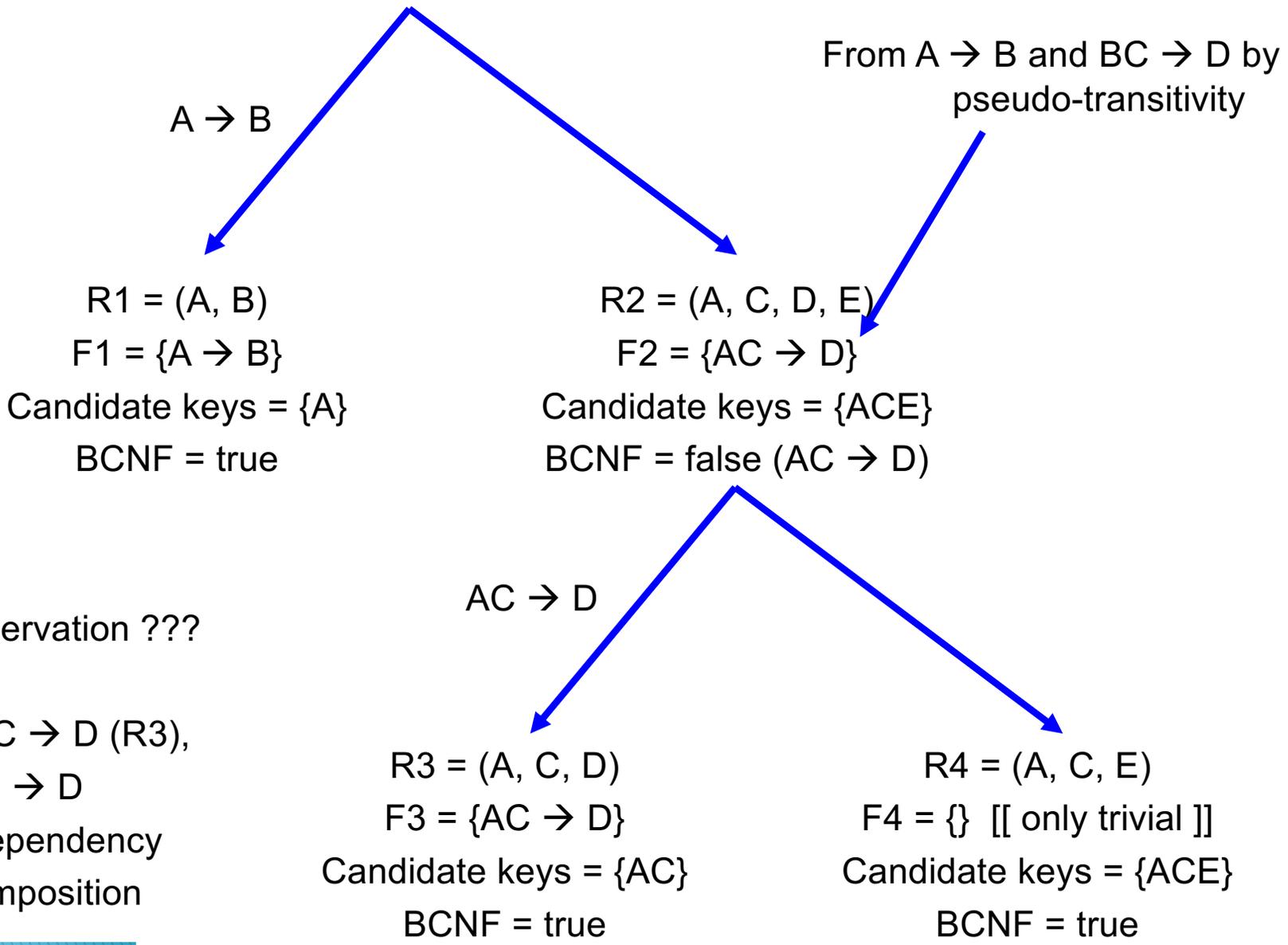
Example 2-1

$R = (A, B, C, D, E)$

$F = \{A \rightarrow B, BC \rightarrow D\}$

Candidate keys = $\{ACE\}$

BCNF = Violated by $\{A \rightarrow B, BC \rightarrow D\}$ etc...



Dependency preservation ???

We can check:

$A \rightarrow B$ (R1), $AC \rightarrow D$ (R3),
but we lost $BC \rightarrow D$

So this is not a dependency-preserving decomposition

Example 2-2

$R = (A, B, C, D, E)$

$F = \{A \rightarrow B, BC \rightarrow D\}$

Candidate keys = $\{ACE\}$

BCNF = Violated by $\{A \rightarrow B, BC \rightarrow D\}$ etc...

$BC \rightarrow D$

$R_1 = (B, C, D)$

$F_1 = \{BC \rightarrow D\}$

Candidate keys = $\{BC\}$

BCNF = true

$R_2 = (B, C, A, E)$

$F_2 = \{A \rightarrow B\}$

Candidate keys = $\{ACE\}$

BCNF = false ($A \rightarrow B$)

$A \rightarrow B$

$R_3 = (A, B)$

$F_3 = \{A \rightarrow B\}$

Candidate keys = $\{A\}$

BCNF = true

$R_4 = (A, C, E)$

$F_4 = \{ \}$ [[only trivial]]

Candidate keys = $\{ACE\}$

BCNF = true

Dependency preservation ???

We can check:

$BC \rightarrow D$ (R_1), $A \rightarrow B$ (R_3),

Dependency-preserving decomposition



Example 3

$R = (A, B, C, D, E, H)$

$F = \{A \rightarrow BC, E \rightarrow HA\}$

Candidate keys = $\{DE\}$

BCNF = Violated by $\{A \rightarrow BC\}$ etc...

$A \rightarrow BC$

$R_1 = (A, B, C)$

$F_1 = \{A \rightarrow BC\}$

Candidate keys = $\{A\}$

BCNF = true

$R_2 = (A, D, E, H)$

$F_2 = \{E \rightarrow HA\}$

Candidate keys = $\{DE\}$

BCNF = false ($E \rightarrow HA$)

$E \rightarrow HA$

$R_3 = (E, H, A)$

$F_3 = \{E \rightarrow HA\}$

Candidate keys = $\{E\}$

BCNF = true

$R_4 = (ED)$

$F_4 = \{ \}$ [[only trivial]]

Candidate keys = $\{DE\}$

BCNF = true

Dependency preservation ???

We can check:

$A \rightarrow BC$ (R_1), $E \rightarrow HA$ (R_3),

Dependency-preserving decomposition



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 may not preserve dependencies

- ▶ $R = (J, K, L)$
 - ▶ $F = \{JK \rightarrow L, L \rightarrow K\}$
 - ▶ Two candidate keys = JK and JL
 - ▶ R is not in BCNF
 - ▶ Any decomposition of R will fail to preserve
 $JK \rightarrow L$
 - ▶ This implies that testing for $JK \rightarrow L$ requires a join
- 

BCNF may not preserve dependencies

- ▶ Not always possible to find a dependency-preserving decomposition that is in BCNF.
 - ▶ PTIME to determine if there exists a dependency-preserving decomposition in BCNF
 - in size of F
 - ▶ NP-Hard to find one if it exists
 - ▶ Better results exist if F satisfies certain properties
- 

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:

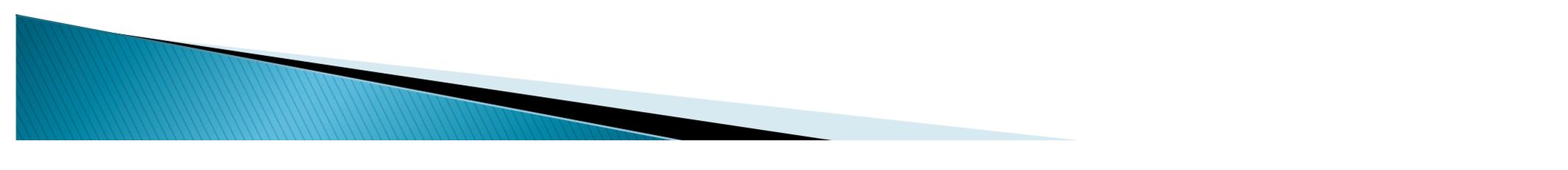
1. A *key* has no redundancy (is not repeated in a relation)
2. A *prime attribute* has limited redundancy

3NF

- ▶ Given a relation schema R , and a set of functional dependencies F , if every FD, $A \rightarrow B$, is either:
 1. Trivial, or
 2. A is a *superkey* of R , or
 3. All attributes in $(B - A)$ are *prime*

Then, R is in *3NF (3rd Normal Form)*

- ▶ Why is 3NF good ?



3NF and Redundancy

▶ Why does redundancy arise ?

- Given a FD, $A \rightarrow B$, if A is repeated (B – A) has to be repeated
 1. If rule 1 is satisfied, (B – A) is empty, so not a problem.
 2. If rule 2 is satisfied, then A can't be repeated, so this doesn't happen either
 3. If not, rule 3 says (B – A) must contain only *prime attributes*
This limits the redundancy somewhat.

▶ So 3NF relaxes BCNF somewhat by allowing for some (hopefully limited) redundancy

▶ Why ?

- *There always exists a dependency-preserving lossless decomposition in 3NF.*

Decomposing into 3NF

- ▶ A *synthesis* algorithm
- ▶ Start with the canonical cover, and construct the 3NF schema directly
- ▶ Homework assignment.



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

MovieTitle	MovieYear	StarName	Address
Star wars	1977	Harrison Ford	Address 1, LA
Star wars	1977	Harrison Ford	Address 2, FL
Indiana Jones	198x	Harrison Ford	Address 1, LA
Indiana Jones	198x	Harrison Ford	Address 2, FL
Witness	19xx	Harrison Ford	Address 1, LA
Witness	19xx	Harrison Ford	Address 2, FL
...

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 \twoheadrightarrow *address*

starname \twoheadrightarrow *movietitle, movieyear*

- ▶ Should not happen if the 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, $A \twoheadrightarrow B$, is either:
 1. Trivial, or
 2. A is a *superkey* of R

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 untrue.

Comparing the normal forms

	3NF	BCNF	4NF
Eliminates redundancy because of FD's	Mostly	Yes	Yes
Eliminates redundancy because of MVD's	No	No	Yes
Preserves FDs	Yes.	Maybe	Maybe
Preserves MVDs	Maybe	Maybe	Maybe

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

- ▶ 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* schema that is then normalized
 - MVDs may be needed in this case



Recap

- ▶ What about 1st and 2nd normal forms ?
- ▶ 1NF:
 - Essentially says that no set-valued attributes allowed
 - Formally, a domain is called *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:
 - 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
 - So 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

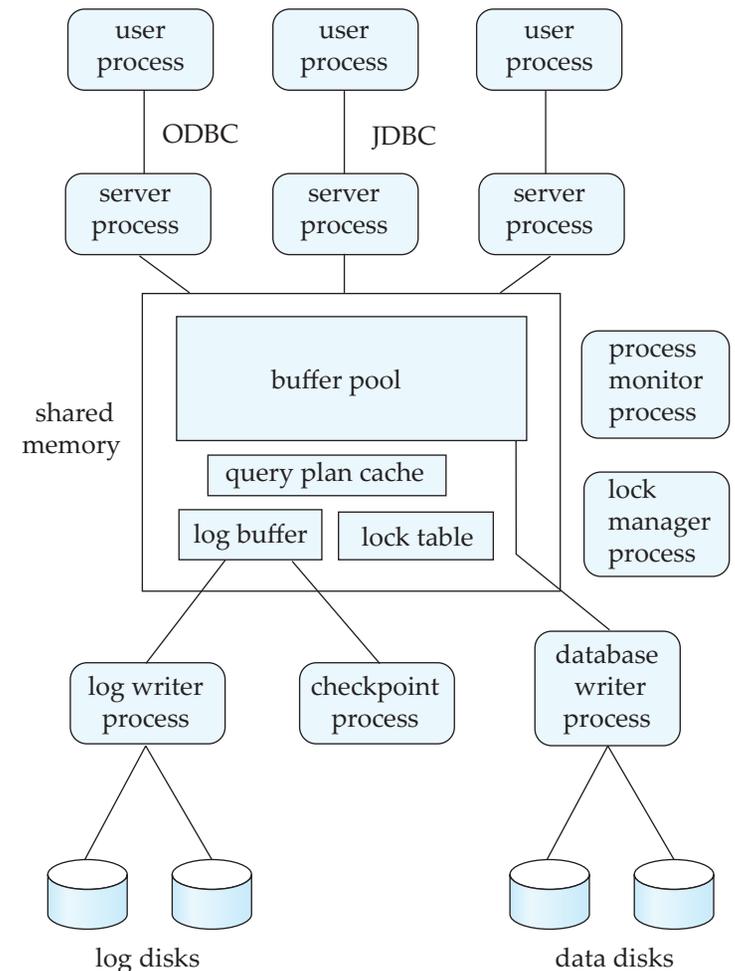


Plan for Today

- ▶ Wrap up Normalization
- ▶ Projects
 - Will start using ELMS for announcements
 - Regrading etc.
- ▶ Midterm 1 on Wednesday: Questions?
- ▶ Next topic:
 - How to "execute" an SQL Query?
 - Today: General background and alternatives

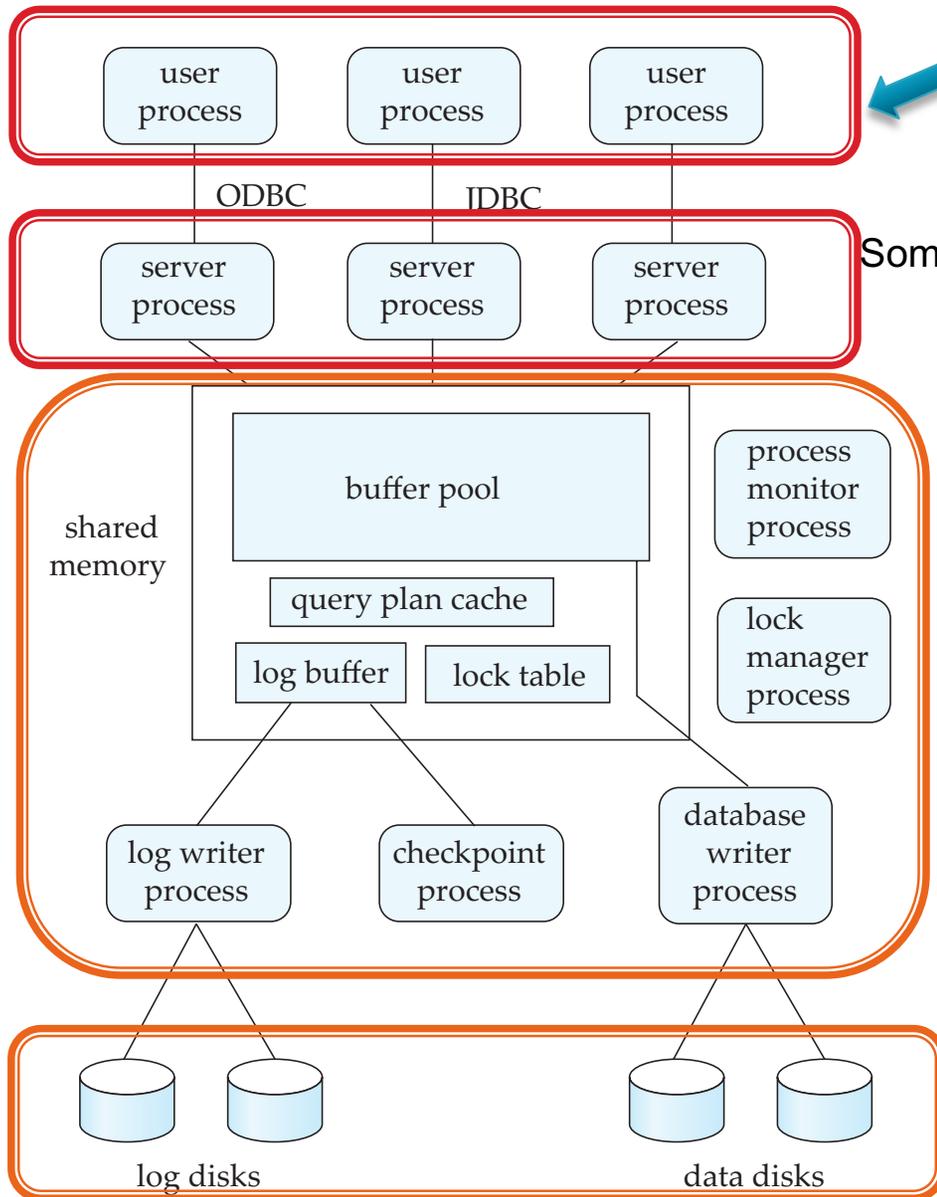
Database Architecture: Pre-2000's

- ▶ All data was typically in hard disks or arrays of hard disks
- ▶ RAM (Memory) was never enough
 - So always had to worry about what was in memory vs not
- ▶ Almost no real “distributed” execution
 - Different from “parallel”, i.e., on co-located clusters of computers
- ▶ Relatively well-understood use cases
 - Report generation
 - Interactive data analysis and exploration
 - Supporting transactions



From Chapter 20

Traditional RDBMS Architecture



Clients may be anywhere – e.g., ATMs, desktops, laptops, web apps etc.

Talk to the database using standard protocols like JDBC/ODBC, SOAP, or REST (today), or proprietary protocols

Some sort of load balancer or intake mechanism

Typical components in a database system: some for queries, some for transactions

Maybe on a single physical computer or a cluster connected by a fast network

Data Storage Systems:

- (1) Punch cards (long time ago)
- (2) Hard disks (still prevalent)
- (3) SSDs

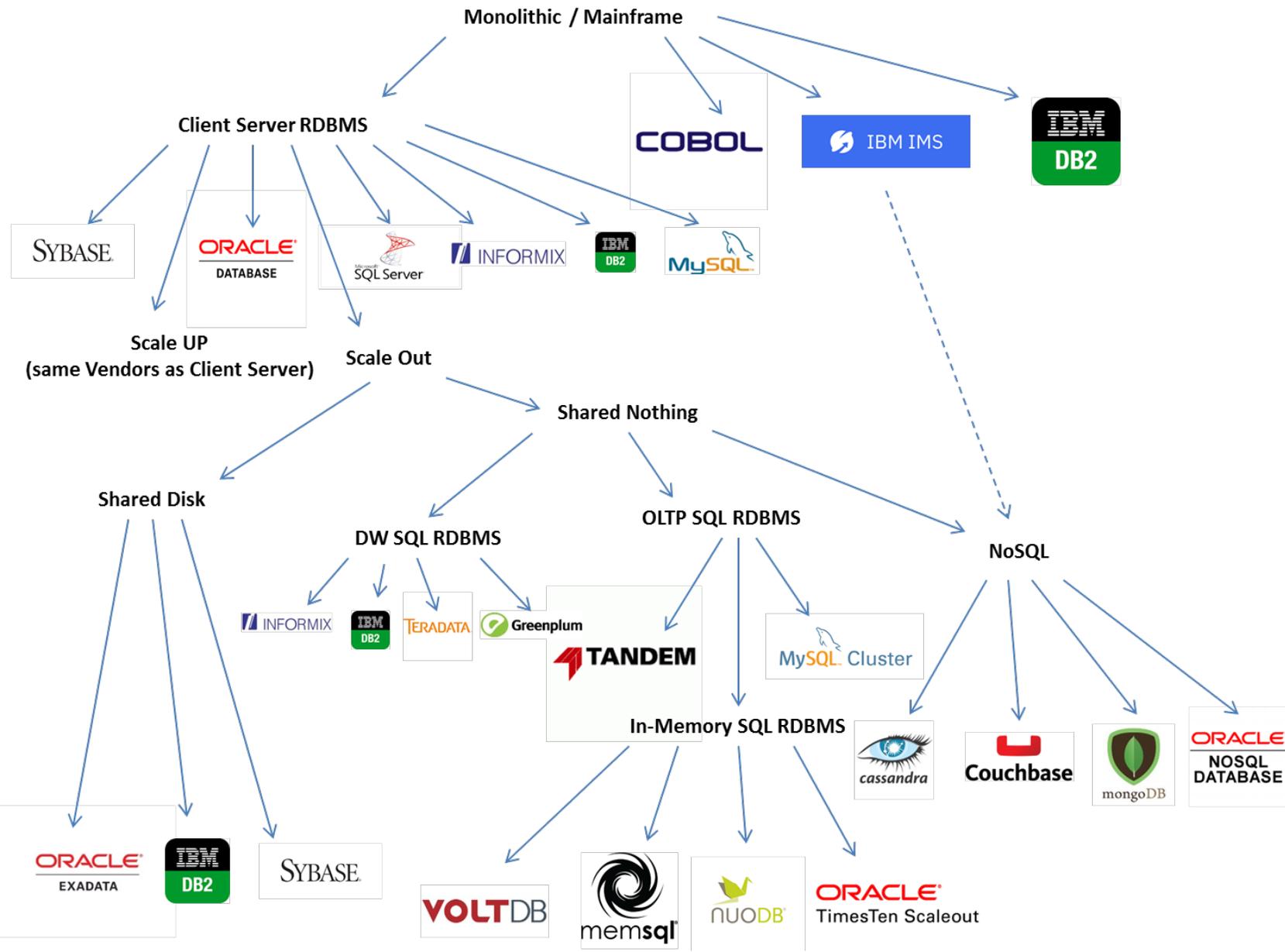
Need “redundancy” and “fault-tolerance”
Data once stored should always be there

RAID = Redundant Array of Independent Disks

Database Architecture : Today

- ▶ **Much more diversity in the architectures that we see**
 - More modern hardware architectures
 - Massively parallel computers
 - SSDs
 - Massive amounts of RAM – often don't need to worry about data fitting in memory
 - Much faster networks, even over a wide area
 - Virtualization and Containerization
 - Cloud Computing
 - As a result: Data and execution typically distributed all over the place
- ▶ **Much more diversity in data processing applications**
 - Much more non-relational data (images, text, video)
 - Data Analytics/Machine learning more common use-cases
- ▶ **Much more diversity in “data models”**
 - Document data models (JSON, XML), Key-value data model, Graph data model, RDF





From: <https://blogs.oracle.com/timesten/the-evolution-of-db-architectures>
 (Oracle-focused)

Data Warehouses

For: Large-scale data processing (TBs to PBs)

Parallel architectures (lots of co-located computers)

SQL and Reporting

No transactions

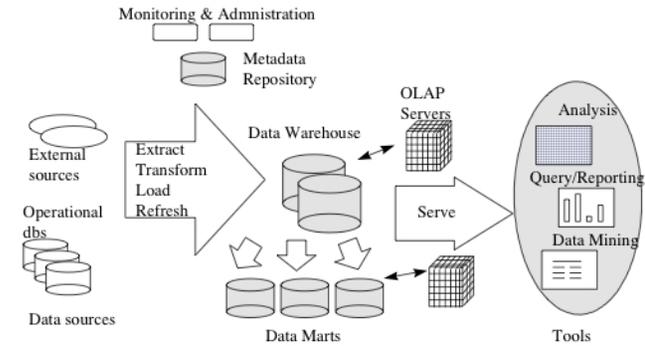


Figure 1. Data Warehousing Architecture

In-memory OLTP (on-line transaction processing)

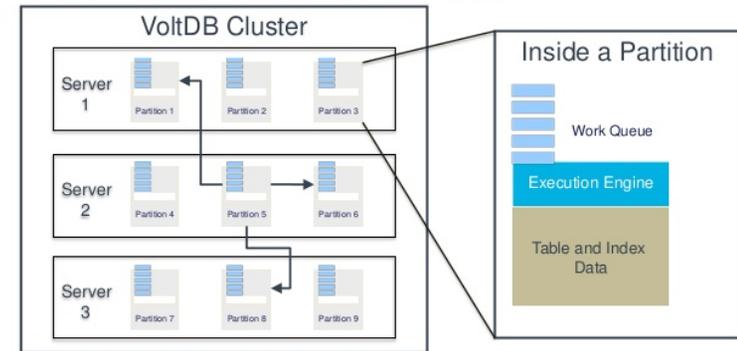
For: Extremely fast transactions

Many-core or parallel architectures

Very limited SQL – mostly focused on “writes”

Typically assume data fits in memory across servers

VOLTDDB: A BEAUTIFUL ARCHITECTURE



Highly available, distributed OLTP

For: Distributed scenarios where clients are all over the world

Focus on “consistency” – how to make sure all users see the same data

Limited SQL – mostly focused on “writes”

Considerations of memory vs disk less important

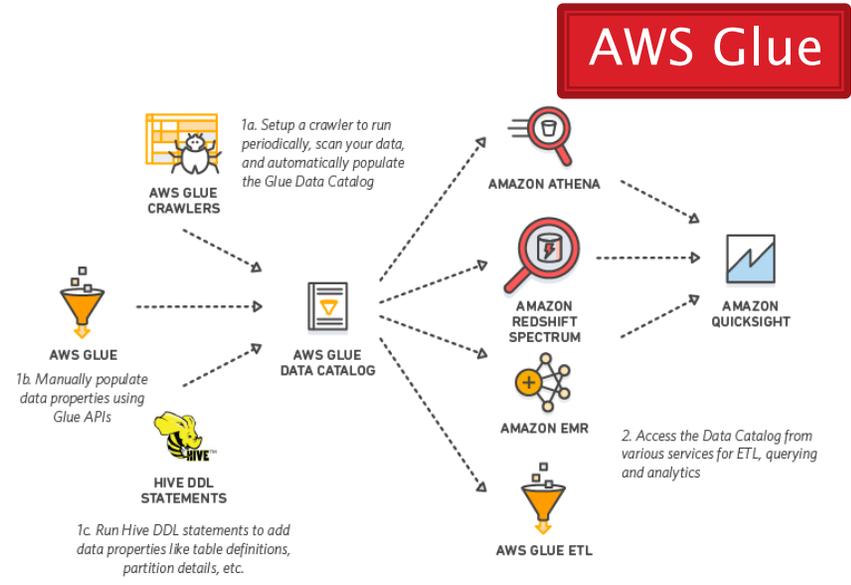


Extract-Transform-Load Systems, or Map-Reduce, or Big Data Frameworks

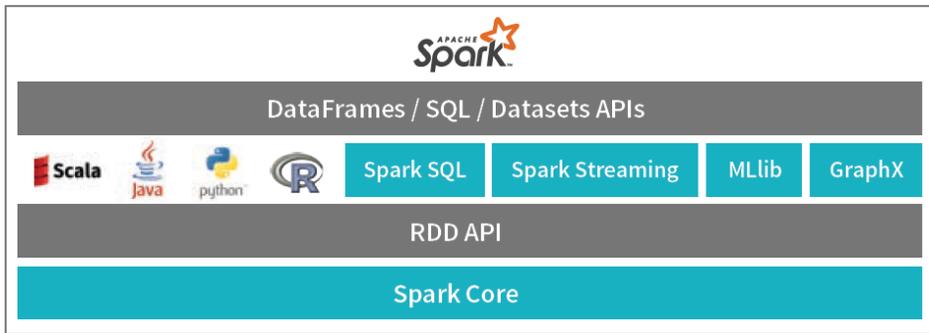
For: Large-scale, "ad hoc" data analysis

Mix of parallel and distributed architectures
Data usually coming from many different sources

Mix of SQL, Machine Learning, and ad hoc tasks (e.g., do image analysis, followed by SQL)



Applications



Apache Spark

Environments

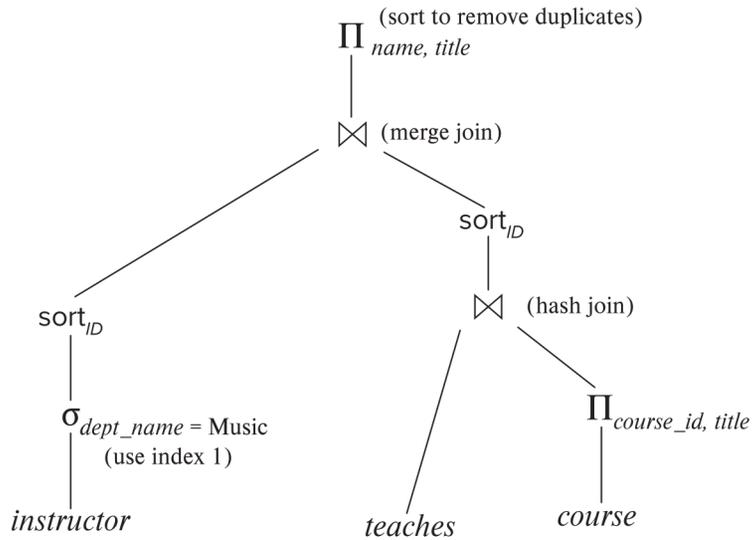


Data Sources

Okay...

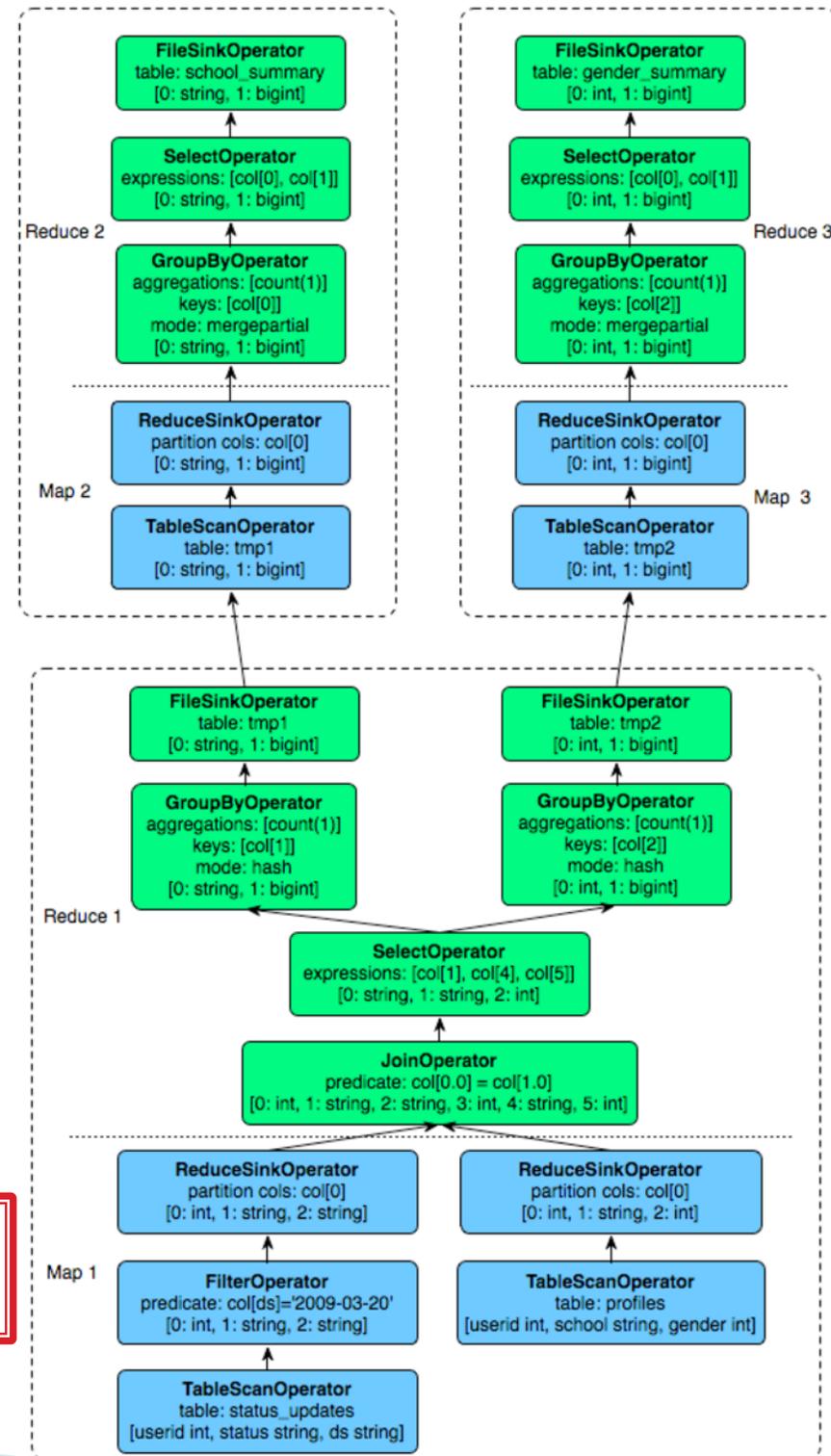
- ▶ Key takeaway: Modern data architectures are a mess
 - We haven't talked about NoSQL (MongoDB, etc.), Machine Learning, "Streaming" ...
 - ▶ Fundamentals haven't changed that much though
 - We are still either:
 - Going from some "input datasets" to an "output dataset" (queries/analytics)
 - Modifying data (transactions)
 - SQL is still very common, albeit often disguised
 - Spark RDD operations map nicely to SQL joins and aggregates (unified now)
 - MongoDB lookups, filters, and aggregates map to joins, selects, and aggregates in SQL
 - ▶ But "performance trade-offs" are all over the place now
 - 30 years ago, we worried a lot about hard disks and things fitting in memory
 - Today, focus more on networks
 - ▶ Focus has shifted to other aspects of data processing pipelines
 - Analytics/Machine learning, data cleaning, statistics
- 

Query Plans vs...

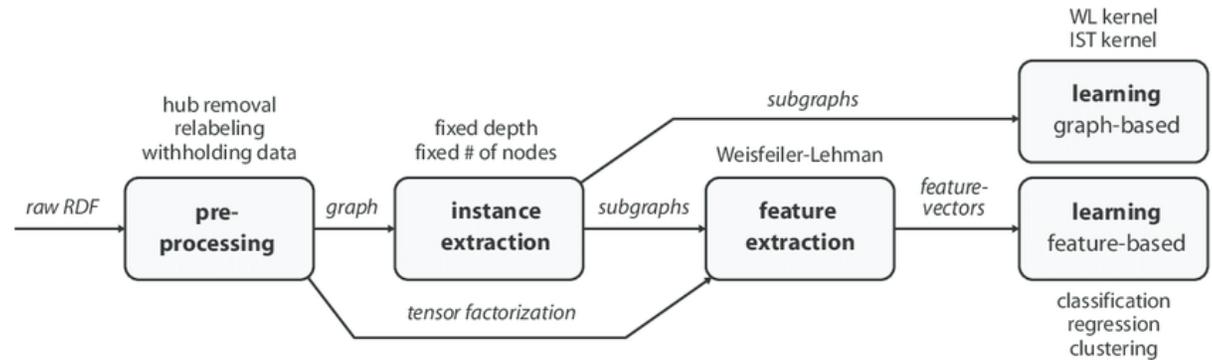


SQL "Query Plan"

Apache Hive "Query Plan"
(Hive is an SQL layer on top of Hadoop)

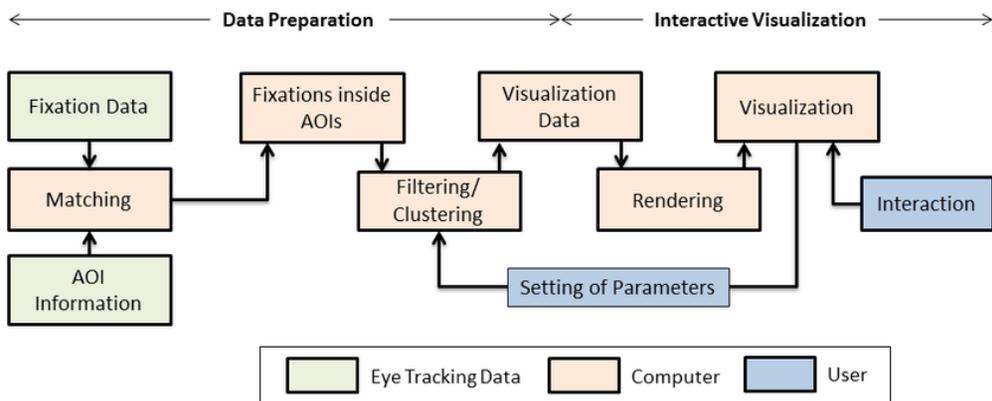


vs ... Data Transformation Pipelines



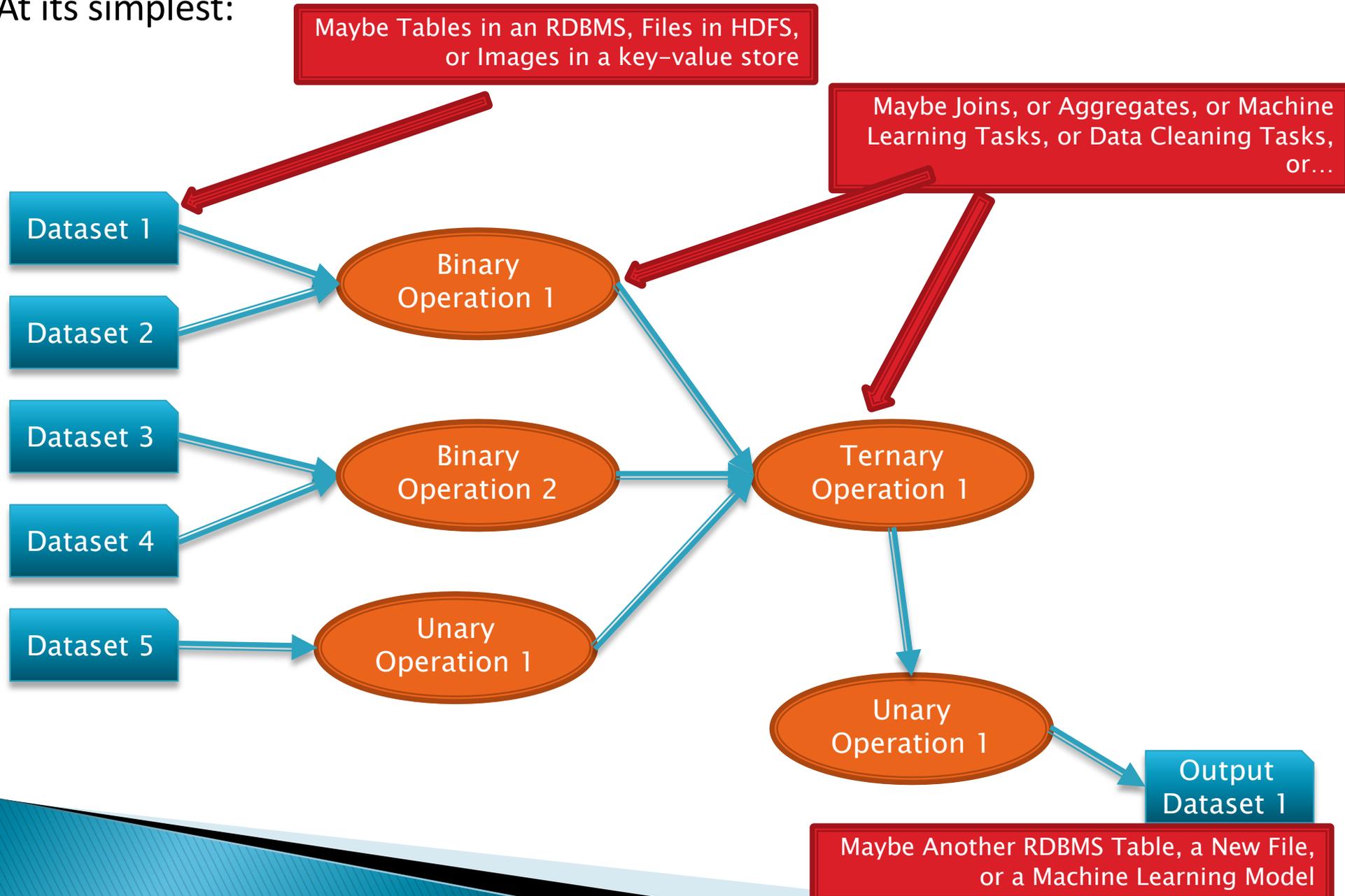
Machine Learning Pipeline

Data Preparation and Visualization Pipeline



Okay...

- ▶ Many similarities across different ways to process and analyze data
- ▶ At its simplest:



Okay...

- ▶ Many similarities across different ways to process and analyze data
 - ▶ Some considerations that we see repeated:
 - Are there multiple ways to accomplish the goals?
 - i.e., are there multiple pipelines or SQL Query Plans that will accomplish the same task
 - How to “enumerate” all of them?
 - i.e., how to automatically come up with all the different options?
 - How to decide which is the “best”?
 - Ideally based on some consideration of total cost (e.g., total CPU time)
 - How to “find” the best plan?
 - Called “query optimization” in databases
 - ▶ RDBMSs have been doing this for 4-5 decades now
 - The classic paper on SQL query optimization is from 1979
 - Outlined the approach still in use today
 - ▶ Same ideas re-discovered repeatedly in other contexts (e.g., Hadoop)
- 

In This Class...

- ▶ We have to limit the scope drastically
- ▶ Focus on:
 - Single-server Relational Databases
 - Assume hard disks are still important and memory is limited
 - Go deep into different ways to execute queries, and find the best queries
- ▶ Will briefly discuss:
 - Parallel architectures and query processing there
 - Map-reduce architectures and considerations there-in
- ▶ Most of the key concepts valid in modern databases (including NoSQL) and Big Data Frameworks

