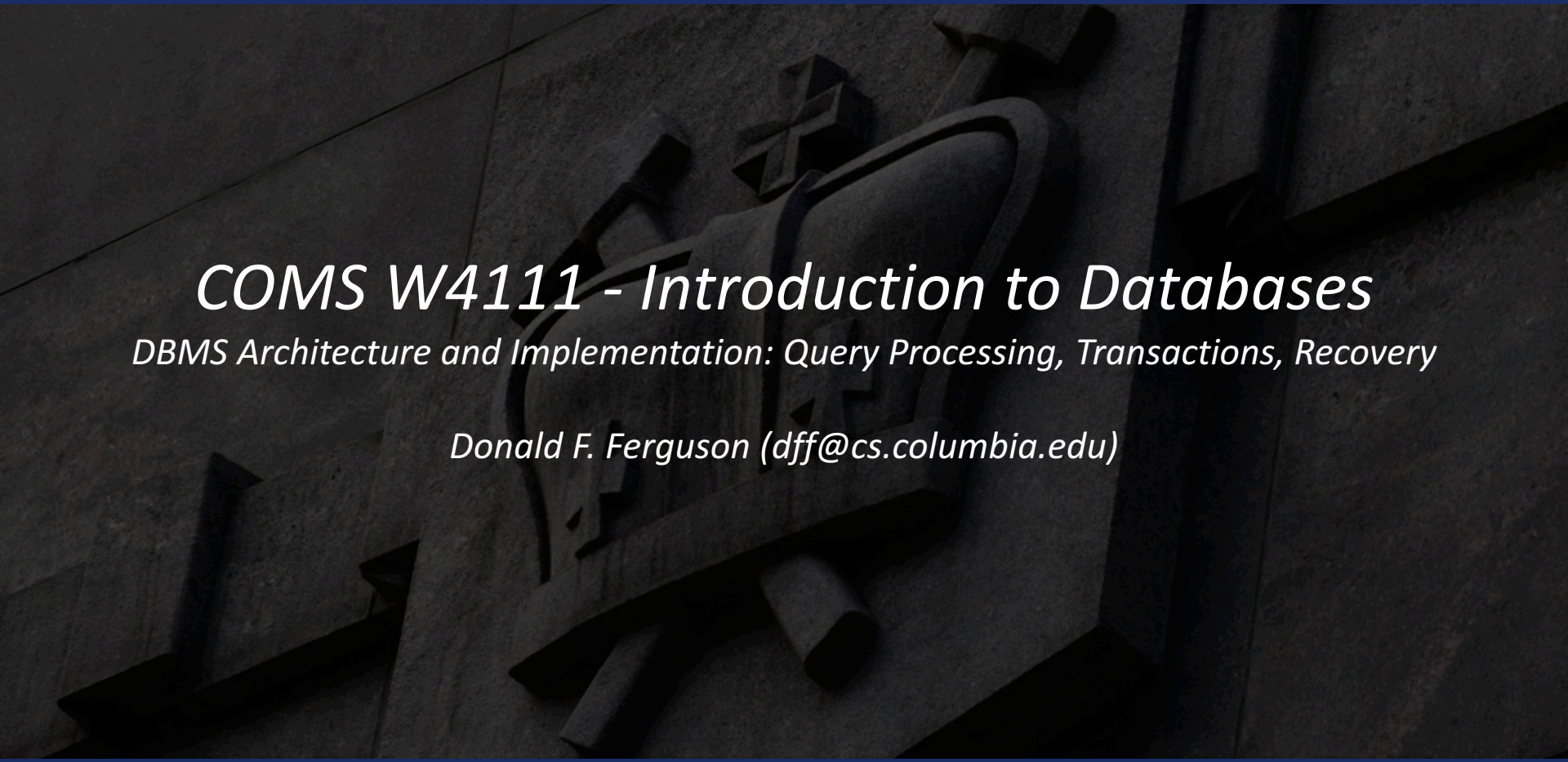


COMS W4111 - Introduction to Databases

DBMS Architecture and Implementation: Query Processing, Transactions, Recovery

Donald F. Ferguson (dff@cs.columbia.edu)

Contents

Contents

- Roadmap for the remainder of the course/semester:
  - Topics
  - Homework
  - Final Exam
- Completing query processing and query optimization:
  - Reminder: Overview and Concepts
  - Some Examples
  - Homework 3
- Transactions: Concepts, Implementation, Examples
- Availability and Recovery: Concepts, ARIES Algorithm

Semester Roadmap

Modules – Reminder

Module I: Foundational Concepts

1. Introduction to databases, role in applications, type of DB applications and overall system software architecture.
2. Information and data modeling and best practices, focusing on supporting application scenarios.
3. Relational data model (theory), Relational Database Management Systems, Structured Query Language, data query and update scenarios.
4. Extended topics in SQL and RDBMS (performance, security, constraints, triggers, connection management, etc).

Module II: Database Management System Implementation/Architecture

5. Storage management, disk management, buffer management, indexes.
6. Query processing and optimization: Query evaluation, query parsing and parse trees, operator implementation algorithms, query rewrite, query optimization techniques.
7. Concurrency control and transaction management.

Module III: NoSQL Database Overview

8. Overview, graph databases, Redis.
9. Amazon S3, Amazon DynamoDB, Google Firebase/Cloud Firestore.

Module IV: Decision Support, Data Analysis

10. Overview of schema denormalization, OLAP cubes, data analytics, machine learning.

Planned Schedule -- Lectures

Lectures

- 09-Nov:
  - Complete query optimization
  - Transactions
  - Availability, Recovery
- 16-Nov: NoSQL Databases
  - Neo4J – Graph database
  - Redis – Key/Value, data structure store
  - Amazon S3 – Blob storage
  - Amazon DynamoDB – Partitioned, JSON
- 30-Nov: Big Data/Decision Support
  - Hadoop/Mapreduce
  - OLAP, Pivot Tables
  - De-normalization
  - Machine Learning
- 07-Dec: Overflow, Final exam discussion

Exams/Assignments

- HW3:
  - Published: 05-Nov
  - Due: 18-Nov
- HW4:
  - Publication: 16-Nov
  - Due: 30-Nov
- HW5:
  - Publication: 30-Nov
  - Due: 10-Dec
- Final Exam:
  - Publication: 07-Dec
  - Due: 16-Dec

Planned Schedule -- Lectures

- I am away 09-Nov – 14-Nov
- I want to make sure I can have enough OH/recitation near due date.

- 16-Nov: NoSQL Databases
  - Neo4J – Graph database
  - ore

- Should be significantly easier and shorter than HW 1, 2 or 3.
- But, …clearly, I am a bad judge of "easy." I will try.
- I want you to get some exposure and be able to put on resume'.

  - Machine Learning
- 07-Dec: Overflow, Final exam discussion

Exams/Assignments

- HW3:
  - Published: 05-Nov
  - Due: 18-Nov
- HW4:
  - Publication: 16-Nov
  - Due: 30-Nov
- HW5:
  - Publication: 30-Nov
  - Due: 10-Dec
- Final Exam:
  - Publication: 07-Dec
  - Due: 16-Dec

Module II – Reminder

Database Management System Reminder

What is a database? In essence a database is nothing more than a collection of information that exists over a long period of time, often many years. In common parlance, the term *database* refers to a collection of data that is managed by a DBMS. The DBMS is expected to:

1. Allow users to create new databases and specify their *schemas* (logical structure of the data), using a specialized *data-definition language*.

Covered for the relational model.

Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Database Management System Reminder

2. Give users the ability to *query* the data (a "query" is database lingo for a question about the data) and modify the data, using an appropriate language, often called a *query language* or *data-manipulation language*.

3. Support the storage of very large amounts of data — many terabytes or more — over a long period of time, allowing efficient access to the data for queries and database modifications.

4. Enable *durability*, the recovery of the database in the face of failures, errors of many kinds, or intentional misuse.

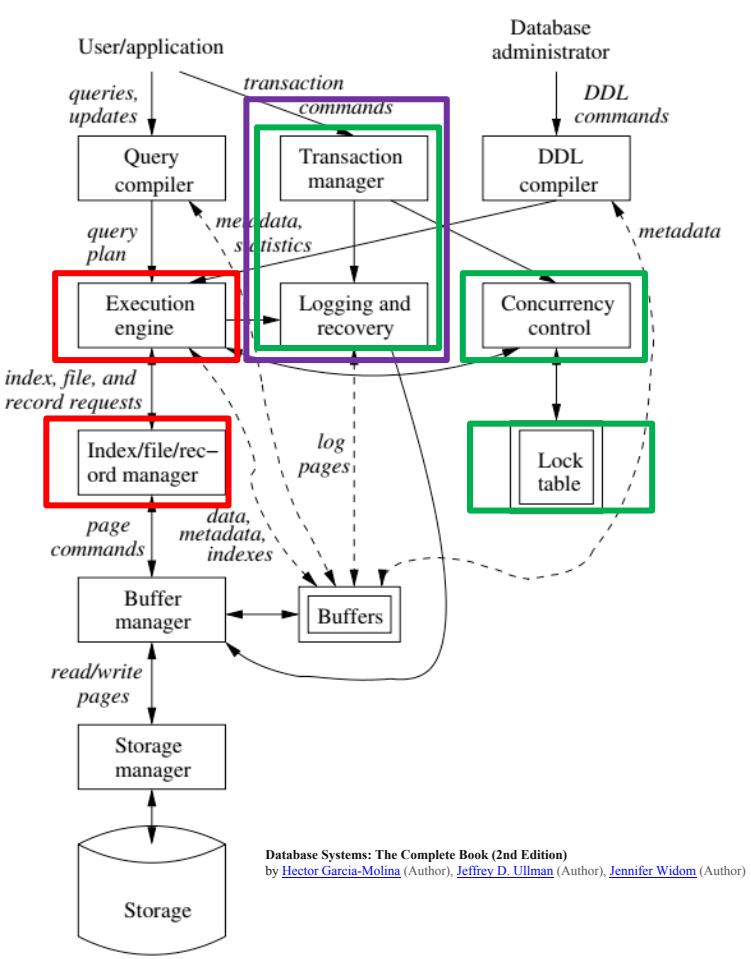
5. Control access to data from many users at once, without allowing unexpected interactions among users (called *isolation*) and without actions on the data to be performed partially but not completely (called *atomicity*).

Focus of current part of course.

Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Data Management

- Find things quickly.
- Control access.
- Durability



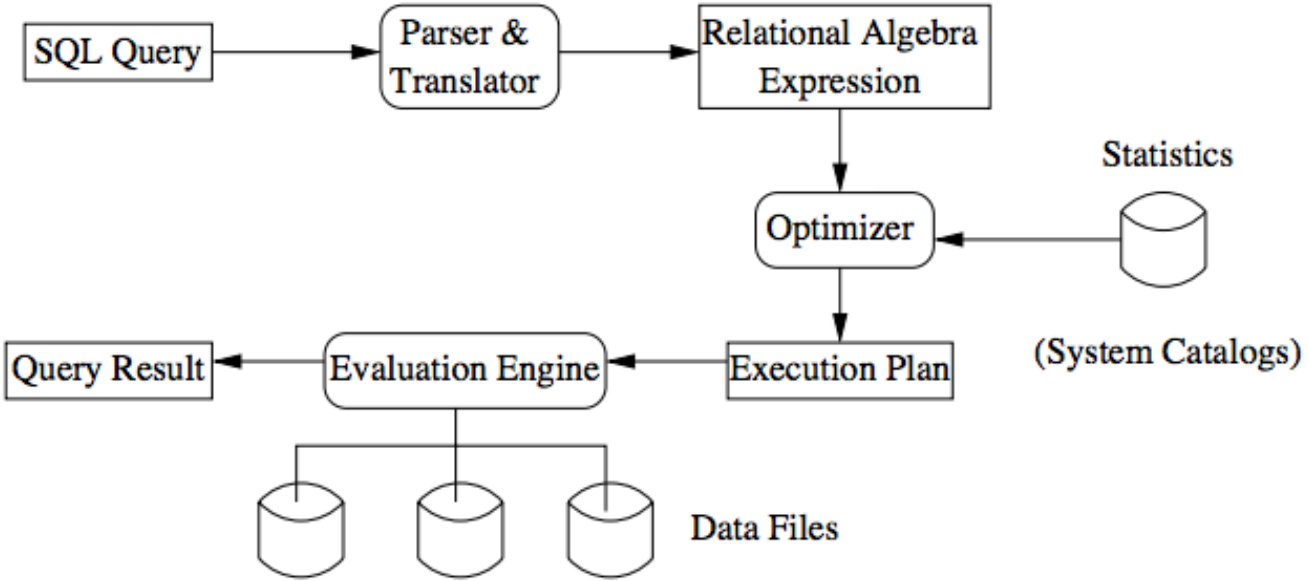
Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Query Processing

Reminder

Query Processing Overview

Basic Steps in Processing an SQL Query



Query Compilation

Database Systems: The Complete Book (2nd Edition) 2nd Edition
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Preview of Query Compilation

To set the context for query execution, we offer a very brief outline of the content of the next chapter. Query compilation is divided into the three major steps shown in Fig. 15.2.

a) *Parsing.* A *parse tree* for the query is constructed.

b) *Query Reurite.* The parse tree is converted to an initial query plan, which is usually an algebraic representation of the query. This initial plan is then transformed into an equivalent plan that is expected to require less time to execute.

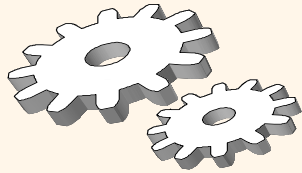
c) *Physical Plan Generation.* The abstract query plan from (b), often called a *logical query plan*, is turned into a *physical query plan* by selecting algorithms to implement each of the operators of the logical plan, and by selecting an order of execution for these operators. The physical plan, like the result of parsing and the logical plan, is represented by an expression tree. The physical plan also includes details such as how the queried relations are accessed, and when and if a relation should be sorted.

Previous Lecture

Current Lecture

Optimization/Execution

Overview of Query Evaluation



- *Plan: Tree of R.A. ops, with choice of alg for each op.*
  - Each operator typically implemented using a `pull' interface: when an operator is `pulled' for the next output tuples, it `pulls' on its inputs and computes them.

- Two main issues in query optimization:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?

- Ideally: Want to find best plan.
  Practically: Avoid worst plans!

Query Optimization

Parts (b) and (c) are often called the *query optimizer*, and these are the hard parts of query compilation. To select the best query plan we need to decide:

1. Which of the algebraically equivalent forms of a query leads to the most efficient algorithm for answering the query?
2. For each operation of the selected form, what algorithm should we use to implement that operation?
3. How should the operations pass data from one to the other, e.g., in a pipelined fashion, in main-memory buffers, or via the disk?

Each of these choices depends on the metadata about the database. Typical metadata that is available to the query optimizer includes: the size of each relation; statistics such as the approximate number and frequency of different values for an attribute; the existence of certain indexes; and the layout of data on disk.

Previous Lecture

Homework 3 – "Diversion"

CREATE TABLE

```
def define_tables():
    """
    This would be done by a DBA. This code is simulating CREATE TABLE statements.
    The underlying CSV file contains the data. HW3 assumes some other code performs insert, update, delete
    on the data files.
    :return: None
    """
    cleanup()
    cat = CSVCatalog.CSVCatalog()                                    # Connect to catalog to perform DDL

    # Define columns.
    cds = []
    cds.append(CSVCatalog.ColumnDefinition("playerID", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("nameLast", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("nameFirst", column_type="text"))
    cds.append(CSVCatalog.ColumnDefinition("birthCity", "text"))
    cds.append(CSVCatalog.ColumnDefinition("birthCountry", "text"))
    cds.append(CSVCatalog.ColumnDefinition("throws", column_type="text"))

    # CREATE TABLE DDL. Put definition information in our version of INFORMATION_SCHEMA.
    t = cat.create_table("people",
        "/Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/Fall2018/W4111/Data/People.csv", cds)

    cds = []
    cds.append(CSVCatalog.ColumnDefinition("playerID", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("H", "number", True))
    cds.append(CSVCatalog.ColumnDefinition("AB", column_type="number"))
    cds.append(CSVCatalog.ColumnDefinition("teamID", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("yearID", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("stint", column_type="number", not_null=True))

    t = cat.create_table("batting",
        "/Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/Fall2018/W4111/Data/Batting.csv", cds)
```

Describe Table

```
def test_describe():

    cat = CSVCatalog.CSVCatalog()
    people_t = cat.get_table("people")
    desc = people_t.describe_table()
    print("DESCRIBE People = \n",
          json.dumps(desc, indent=2))
```

```
DESCRIBE People =
 {
  "definition": {
    "name": "people",
    "path": "/Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/F
  },
  "columns": [
    {
      "column_name": "birthCity",
      "column_type": "text",
      "not_null": false
    },
    {
      "column_name": "birthCountry",
      "column_type": "text",
      "not_null": false
    },
    {
      "column_name": "nameFirst",
      "column_type": "text",
      "not_null": false
    },
    {
      "column_name": "nameLast",
      "column_type": "text",
      "not_null": true
    },
    {
      "column_name": "playerID",
      "column_type": "text",
      "not_null": true
    },
    {
      "column_name": "throws",
      "column_type": "text",
      "not_null": false
    }
  ],
  "indexes": {}
}
```

Analogy to RDB/MySQL

- At some point, DBA defines tables
- Later (much later), programmer can call DESCRIBE TABLE
- The query processing engine will also use this information to
  - Validate statement correctness.
  - Perform optimizations.
- For HW3, we are not worrying about insert, update, delete into the files.

```
CREATE TABLE `W4111`.`people` (
  `playerID` TEXT NOT NULL,
  `nameLast` TEXT NOT NULL,
  `nameFirst` TEXT NULL,
  `birthCity` TEXT NULL,
  `birthCountry` TEXT NULL,
  `throws` TEXT NULL);
```

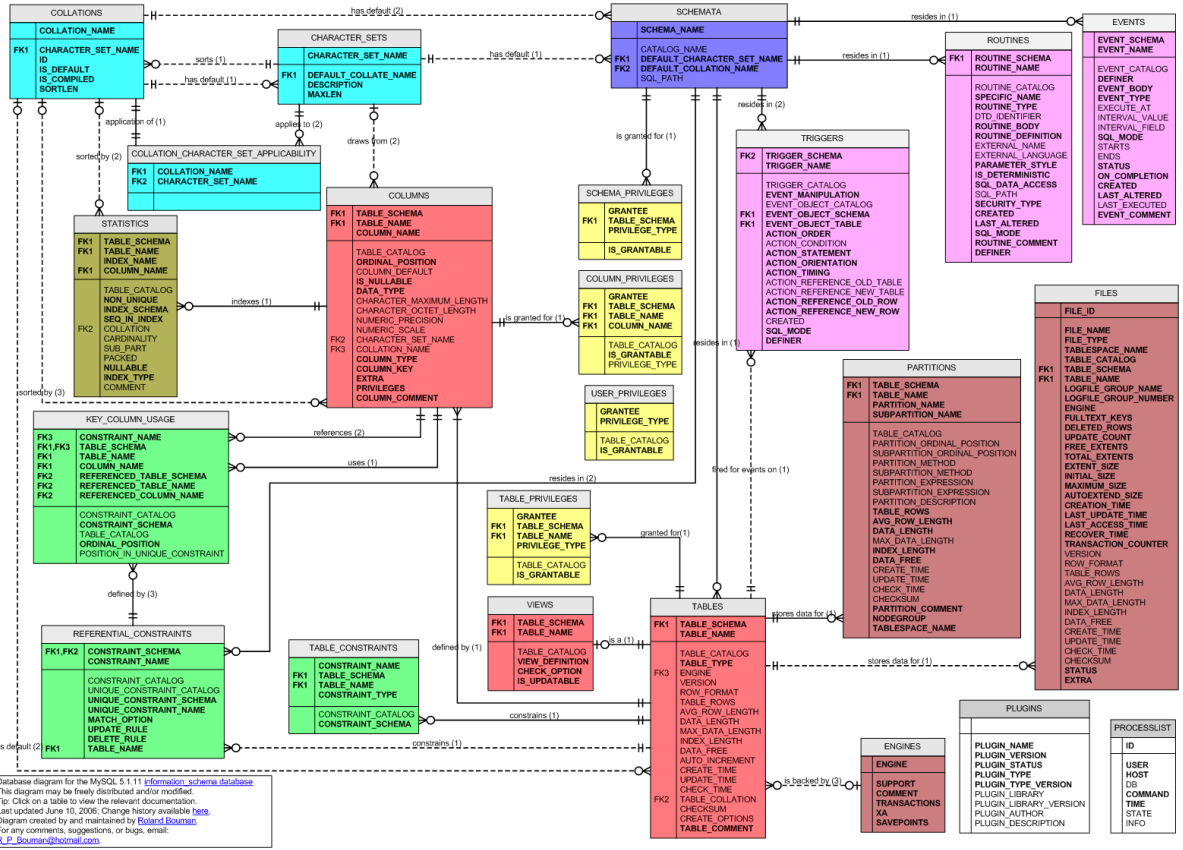
```
1 • describe people
```

16:1

Result Grid | Filter Rows: Search | Export:

Field	Type	Null	Key	Default	Extra
playerID	text	NO		NULL	
nameLast	text	NO		NULL	
nameFirst	text	YES		NULL	
birthCity	text	YES		NULL	
birthCountry	text	YES		NULL	
throws	text	YES		NULL	

MySQL Information Schema



You are doing something

- Similar
- Much simpler
- For your CSVTables

INFORMATION_SCHEMA.tables

```
1 • select * from information_schema.tables where table_schema = 'lahman2017'
```

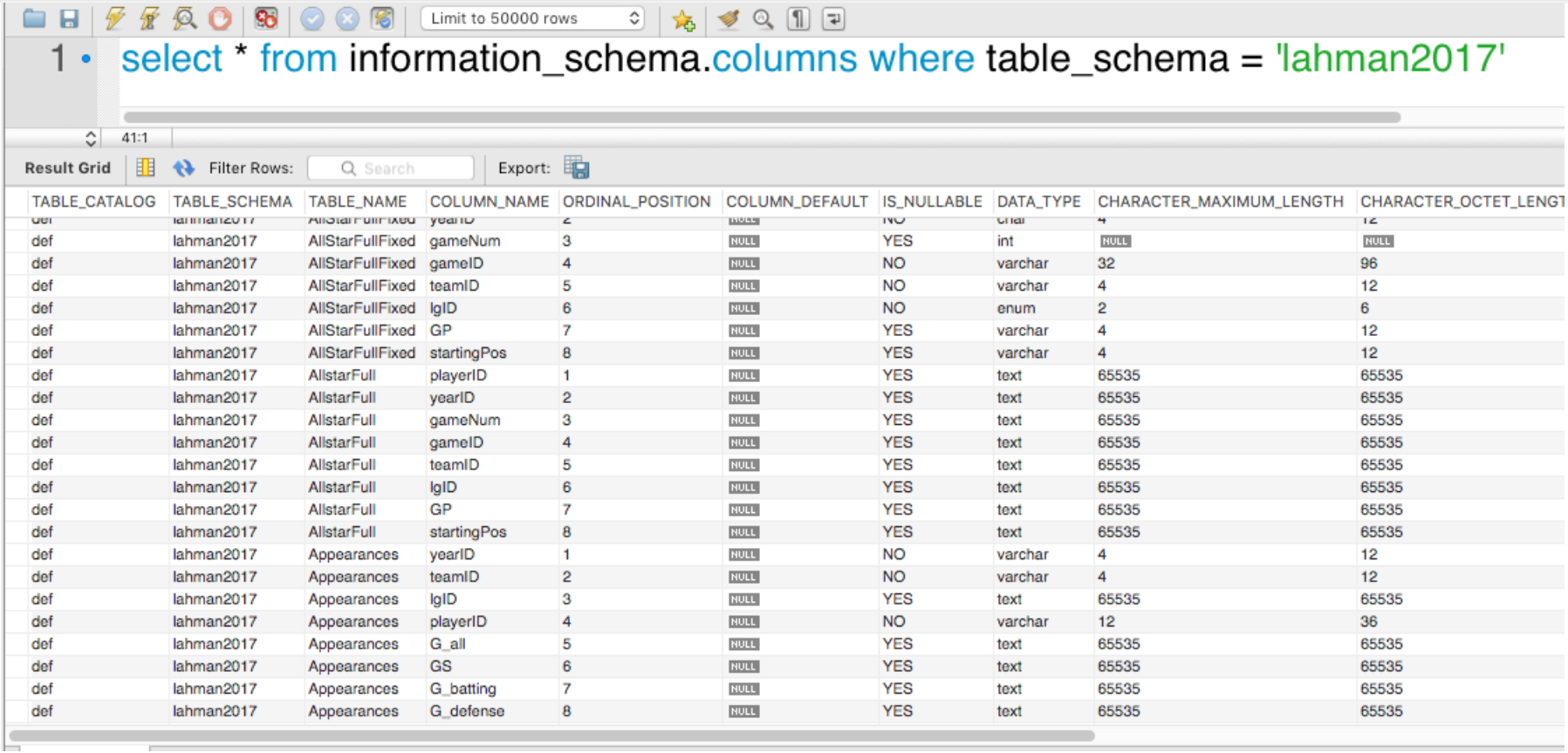
Limit to 50000 rows

63:1

Result Grid | Filter Rows: Search | Export:

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	TABLE_TYPE	ENGINE	VERSION	ROW_FORMAT	TABLE_ROWS	AVG_ROW_LENGTH	DATA_LENGTH	MAX_DATA_LENGTH	INDEX_LENGTH	DATA_FREE	AUTO_INCR
def	lahman2017	AllStarFullFixed	BASE TABLE	InnoDB	10	Dynamic	5219	103	540672	0	0	0	NULL
def	lahman2017	AllstarFull	BASE TABLE	InnoDB	10	Dynamic	5396	75	409600	0	0	0	NULL
def	lahman2017	Appearances	BASE TABLE	InnoDB	10	Dynamic	103464	96	10010624	0	7372800	4194304	NULL
def	lahman2017	AwardsManagers	BASE TABLE	InnoDB	10	Dynamic	179	91	16384	0	0	0	NULL
def	lahman2017	AwardsPlayers	BASE TABLE	InnoDB	10	Dynamic	6326	75	475136	0	0	0	NULL
def	lahman2017	AwardsShareManagers	BASE TABLE	InnoDB	10	Dynamic	425	154	65536	0	0	0	NULL
def	lahman2017	AwardsSharePlayers	BASE TABLE	InnoDB	10	Dynamic	6845	67	458752	0	0	0	NULL
def	lahman2017	Batting	BASE TABLE	InnoDB	10	Dynamic	104218	106	11059200	0	7372800	2097152	NULL
def	lahman2017	BattingPost	BASE TABLE	InnoDB	10	Dynamic	14030	113	1589248	0	0	0	NULL
def	lahman2017	CollegePlaying	BASE TABLE	InnoDB	10	Dynamic	17256	92	1589248	0	0	0	NULL
def	lahman2017	Fielding	BASE TABLE	InnoDB	10	Dynamic	138663	79	11059200	0	7880704	0	NULL
def	lahman2017	FieldingOF	BASE TABLE	InnoDB	10	Dynamic	11980	132	1589248	0	0	0	NULL
def	lahman2017	FieldingOFsplit	BASE TABLE	InnoDB	10	Dynamic	32013	115	3686400	0	0	0	NULL
def	lahman2017	FieldingPost	BASE TABLE	InnoDB	10	Dynamic	13297	119	1589248	0	0	0	NULL
def	lahman2017	HallOfFame	BASE TABLE	InnoDB	10	Dynamic	4191	74	311296	0	147456	0	NULL
def	lahman2017	HomeGames	BASE TABLE	InnoDB	10	Dynamic	3040	97	294912	0	0	0	NULL
def	lahman2017	Managers	BASE TABLE	InnoDB	10	Dynamic	3469	70	245760	0	0	0	NULL
def	lahman2017	ManagersHalf	BASE TABLE	InnoDB	10	Dynamic	93	176	16384	0	0	0	NULL
def	lahman2017	Parks	BASE TABLE	InnoDB	10	Dynamic	252	195	49152	0	0	0	NULL
def	lahman2017	People	BASE TABLE	InnoDB	10	Dynamic	19300	191	3686400	0	0	3145728	NULL
def	lahman2017	Pitching	BASE TABLE	InnoDB	10	Dynamic	44452	130	5783552	0	2637824	0	NULL
def	lahman2017	PitchingPost	BASE TABLE	InnoDB	10	Dynamic	5444	291	1589248	0	0	0	NULL
def	lahman2017	Salaries	BASE TABLE	InnoDB	10	Dynamic	30585	86	2637824	0	0	0	NULL

INFORMATION_SCHEMA.columns



Limit to 50000 rows

```
1 • select * from information_schema.columns where table_schema = 'lahman2017'
```

41:1

Result Grid | Filter Rows: Search | Export:

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	CHARACTER_MAXIMUM_LENGTH	CHARACTER_OCTET_LENGT
def	lahman2017	AllStarFullFixed	gameNum	3	NULL	YES	int	NULL	NULL
def	lahman2017	AllStarFullFixed	gameID	4	NULL	NO	varchar	32	96
def	lahman2017	AllStarFullFixed	teamID	5	NULL	NO	varchar	4	12
def	lahman2017	AllStarFullFixed	lgID	6	NULL	NO	enum	2	6
def	lahman2017	AllStarFullFixed	GP	7	NULL	YES	varchar	4	12
def	lahman2017	AllStarFullFixed	startingPos	8	NULL	YES	varchar	4	12
def	lahman2017	AllstarFull	playerID	1	NULL	YES	text	65535	65535
def	lahman2017	AllstarFull	yearID	2	NULL	YES	text	65535	65535
def	lahman2017	AllstarFull	gameNum	3	NULL	YES	text	65535	65535
def	lahman2017	AllstarFull	gameID	4	NULL	YES	text	65535	65535
def	lahman2017	AllstarFull	teamID	5	NULL	YES	text	65535	65535
def	lahman2017	AllstarFull	lgID	6	NULL	YES	text	65535	65535
def	lahman2017	AllstarFull	GP	7	NULL	YES	text	65535	65535
def	lahman2017	AllstarFull	startingPos	8	NULL	YES	text	65535	65535
def	lahman2017	Appearances	yearID	1	NULL	NO	varchar	4	12
def	lahman2017	Appearances	teamID	2	NULL	NO	varchar	4	12
def	lahman2017	Appearances	lgID	3	NULL	YES	text	65535	65535
def	lahman2017	Appearances	playerID	4	NULL	NO	varchar	12	36
def	lahman2017	Appearances	G_all	5	NULL	YES	text	65535	65535
def	lahman2017	Appearances	GS	6	NULL	YES	text	65535	65535
def	lahman2017	Appearances	G_batting	7	NULL	YES	text	65535	65535
def	lahman2017	Appearances	G_defense	8	NULL	YES	text	65535	65535

INFORMATION_SCHEMA.key_column_usage

```
1 • select * from information_schema.key_column_usage where table_schema = 'lahman2017';
```

50:1

Result Grid | Filter Rows: Search | Export:

CONSTRAINT_CATALOG	CONSTRAINT_SCHEMA	CONSTRAINT_NAME	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	POSITION_IN_UNIQUE_CONSTRAINT	REFERENCED_TABLE_SCH
def	lahman2017	PRIMARY	def	lahman2017	AllStarFullFixed	playerID	1	NULL	NULL
def	lahman2017	PRIMARY	def	lahman2017	AllStarFullFixed	gameID	2	NULL	NULL
def	lahman2017	PRIMARY	def	lahman2017	Appearances	playerID	1	NULL	NULL
def	lahman2017	PRIMARY	def	lahman2017	Appearances	teamID	2	NULL	NULL
def	lahman2017	PRIMARY	def	lahman2017	Appearances	yearID	3	NULL	NULL
def	lahman2017	a_to_p	def	lahman2017	Appearances	playerID	1	1	lahman2017
def	lahman2017	a_to_t	def	lahman2017	Appearances	teamID	1	1	lahman2017
def	lahman2017	a_to_t	def	lahman2017	Appearances	yearID	2	2	lahman2017
def	lahman2017	PRIMARY	def	lahman2017	Batting	playerID	1	NULL	NULL
def	lahman2017	PRIMARY	def	lahman2017	Batting	teamID	2	NULL	NULL
def	lahman2017	PRIMARY	def	lahman2017	Batting	yearID	3	NULL	NULL
def	lahman2017	PRIMARY	def	lahman2017	Batting	stint	4	NULL	NULL
def	lahman2017	b_to_p	def	lahman2017	Batting	playerID	1	1	lahman2017
def	lahman2017	PRIMARY	def	lahman2017	Fielding	playerID	1	NULL	NULL
def	lahman2017	PRIMARY	def	lahman2017	Fielding	teamID	2	NULL	NULL
def	lahman2017	PRIMARY	def	lahman2017	Fielding	yearID	3	NULL	NULL
def	lahman2017	PRIMARY	def	lahman2017	Fielding	stint	4	NULL	NULL
def	lahman2017	PRIMARY	def	lahman2017	Fielding	POS	5	NULL	NULL
def	lahman2017	f_to_p	def	lahman2017	Fielding	playerID	1	1	lahman2017
def	lahman2017	f_to_t	def	lahman2017	Fielding	teamID	1	1	lahman2017
def	lahman2017	f_to_t	def	lahman2017	Fielding	yearID	2	2	lahman2017
def	lahman2017	PRIMARY	def	lahman2017	HallOfFame	playerID	1	NULL	NULL
def	lahman2017	PRIMARY	def	lahman2017	HallOfFame	yearid	2	NULL	NULL

Where is the Data?

```
mysql — jupyter_mac.command — -bash — 94×24
Donalds-MacBook-Pro:mysql donaldferguson$ pwd
/usr/local/mysql
Donalds-MacBook-Pro:mysql donaldferguson$
Donalds-MacBook-Pro:mysql donaldferguson$ sudo ls data
Password:
CSVCatalog              dff_midterm             midtermq4
E1102                   employees               moneyball
E6156                   extracredit             mysql
HW3                     finalexamhw5            mysqld.local.err
JROSS                   ib_buffer_pool          mysqld.local.pid
Midterm                 ib_logfile0             northwind
NYGuard                 ib_logfile1             performance_schema
NewUniversity           ibdata1                 sakila
NewUniversity2          ibtmp1                  sys
Test                    lahman2016              test2
Towns                   lahman2017              transactions
University              lahman2017Raw           university2
W4111                   lahmanslow              useful_data
W4111@002dFinal         midterm3                webappoverview
auto.cnf                midterm@002dorders      webshop
Donalds-MacBook-Pro:mysql donaldferguson$
```

Where is the Data?

```
Donalds-MacBook-Pro:mysql donaldferguson$ sudo ls data/lahman2017
AllStarFullFixed.frm                    batting_y_summary.frm
AllStarFullFixed_BEFORE_INSERT.TRN      batting_year.frm
AllstarFull.frm                         battingpost.ibd
AllstarFull_BEFORE_INSERT.TRN           career_batting.frm
Appearances.frm                         collegeplaying.ibd
AwardsManagers.frm                      copy_tables_are_awesome.frm
AwardsPlayers.frm                       copy_tables_are_awesome.ibd
AwardsShareManagers.frm                 db.opt
AwardsSharePlayers.frm                  fielding.TRG
Batting.frm                             fielding.ibd
BattingPost.frm                         fielding_summary.frm
CollegePlaying.frm                      fielding_y_summary.frm
Fielding.frm                            fielding_year.frm
FieldingOF.frm                          fieldingof.ibd
FieldingOFsplit.frm                     fieldingofsplit.ibd
FieldingPost.frm                        fieldingpost.ibd
Fielding_BEFORE_INSERT.TRN              full_player_year_summary.frm
Fielding_BEFORE_UPDATE.TRN              halloffame.ibd
HallOfFame.frm                          homegames.ibd
HomeGames.frm                           managers.ibd
Managers.frm                            managershalf.ibd
ManagersHalf.frm                        parks.ibd
Parks.frm                               people.ibd
People.frm                              people_summary.frm
Pitching.frm                            people_summary_dff.frm
PitchingPost.frm                        person_three.frm
Salaries.frm                            person_three.ibd
Schools.frm                             pitching.ibd
```

- On my Mac
  - /usr/local/mysql/data
  - /usr/local/mysql/data/lahman2017
    - People.ibd
    - Batting.ibd
    - … …
- For the CSVTables
  - ../Data
    - People.csv
    - Batting.csv
    - … …

The .ibd File

"By default, all InnoDB tables and indexes are stored in the system tablespace. As an alternative, you can store each InnoDB table and associated indexes in its own data**file**. This feature is called "**file**-per-table tablespaces" because each table has its own tablespace, and each tablespace has its own .**ibd** data **file**."

The .ibd File

- MySQL has several approaches to saving table data and index entries.
- .idb/file per table stores uses one file for
  - Tuples
  - Index entries
- Your solution uses
  - One file per table for data.
  - Builds/rebuilds the indexes in memory on load.

Query Optimization – Access Path

Simple "Select" Operation – HW 3

```
def select_slow(tries):

    people_tbl = CSVTable.CSVTable("people")
    template = { "nameLast": "Williams", "birthCity": "San Diego"}
    start_time = time.time()
    for i in range(0, tries):
        result = people_tbl.find_by_template(template, fields=['playerID', 'nameLast', 'birthCity', 'cat'])
        if i == 0:
            print("Testing result. Result = \n", json.dumps(result, indent=2))
    end_time = time.time()
    print("\nElapsed time to execute", tries, "queries = ", end_time-start_time)
```

```
Traceback (most recent call last):
  File "/Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/Fall2018/W4111-Projects/HW3_new/src/HW3_join_test.py", line 83, in <module>
    select_slow(100)
  File "/Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/Fall2018/W4111-Projects/HW3_new/src/HW3_join_test.py", line 73, in select_slow
    result = people_tbl.find_by_template(template, fields="playerID,nameLast,birthCity")
  File "/Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/Fall2018/W4111-Projects/HW3_new/src/CSVTable.py", line 587, in find_by_template
    return self.__find_by_template_scan__(t, fields=fields, limit=None, offset=None)
  File "/Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/Fall2018/W4111-Projects/HW3_new/src/CSVTable.py", line 352, in __find_by_template_scan__
    result = self.project(result, fields)
  File "/Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/Fall2018/W4111-Projects/HW3_new/src/CSVTable.py", line 578, in project
    raise DataTableExceptions.DataTableException(-2, "Invalid field in project")
src.DataTableExceptions.DataTableException: DataTableException: code: -2   , message: Invalid field in project
```

- That is one purpose of the metadata.
- The *find_by_template()* is syntactically correct, but references an invalid column.

Baseline Performance

```
def test_select(tries):

    people_tbl = CSVTable.CSVTable("people")
    template = { "nameLast": "Williams", "birthCity": "San Diego"}
    start_time = time.time()
    for i in range(0, tries):
        result = people_tbl.find_by_template(template, fields=['playerID', 'nameLast', 'birthCity'])
        if i == 0:
            print("Testing result. Result = \n", json.dumps(result, indent=2))
    end_time = time.time()
    print("\nElapsed time to execute", tries, "queries = ", end_time-start_time)
```

```
Testing result. Result =
 [
  {
    "playerID": "willite01",
    "nameLast": "Williams",
    "birthCity": "San Diego"
  },
  {
    "playerID": "willitr01",
    "nameLast": "Williams",
    "birthCity": "San Diego"
  }
]

Elapsed time to execute 1000 queries =  12.48873496055603
```

ALTER TABLE ADD INDEX

```
def test_add_index():
    cat = CSVCatalog.CSVCatalog()
    people_def = cat.get_table("people")
    people_def.define_index('ln_idx', 'INDEX', ['nameLast'])
```

- Adding a non-unique index on “nameLast”
- Improves performance by three orders of magnitude.

```
Testing result. Result =
 [
  {
    "playerID": "willite01",
    "nameLast": "Williams",
    "birthCity": "San Diego"
  },
  {
    "playerID": "willitr01",
    "nameLast": "Williams",
    "birthCity": "San Diego"
  }
]

Elapsed time to execute 1000 queries =  0.06574583053588867
```

find_by_template()

```
def find_by_template(self, t, fields=None, limit=None, offset=None):
    if t is not None:
        access_index, count = self.__get_access_path__(list(t.keys()))
    else:
        access_index = None

    if access_index is None:
        return self.__find_by_template_scan__(t, fields=fields, limit=None, offset=None)
    else:
        result = self.__find_by_template_index__(t, access_index, fields, limit, offset)
        return result
```

- Use the WHERE clause to find the “best” matching index.
- If there is an index, use it.
- If not, must scan the table.

Access Path

- Every relational operator accepts one or more tables as input.
  The operator "accesses the tuples" in the tables.
- There are two "ways" to retrieve the tuples
  - Scan the relation (via blocks)
  - Use an index and matching condition.
- A selection (WHERE) condition is in *conjuctive normal form* if
  - Each term is column_name op value
  - op is one of <, <=, =, >, >=, <> (or =, != for Hash Index)
  - The terms are ANDed, e.g. (nameLast = 'Ferguson') AND (ab > 500)
  - CNF allows using a matching index for the access path.
- The engine can select a *matching index*
  - A Hash Index on (c1,c2,c3) if the condition is c1=x AND c2=y AND c3=z.
  - A Tree Index if the condition is (can be ordered as)
    - c1 op value
    - c1 op x AND/OR c2 op y
    - … …
  - Many indexes may match, and the engine chooses the *most selective match* (number of tuples or blocks) that match.

Access Path

- Every relational operator accepts one or more tables as input.
- The
- T
- A
- T

- If the condition is (nameLast = ‘Williams) AND (birthCity = ‘San Diego’)
  - Use the index to find entries with nameLast ‘Williams’
  - Scan the (smaller) result to find tuples with birthCity=‘San Diego’
- If the condition is (nameLast = ‘Williams) OR (birthCity = ‘San Diego’)
  - The index does not help.
  - I could use to find “Williams”
  - But need to scan anyway for ‘birthCity.’

Query Optimization – Overview

Three Core Optimization Techniques

- Query rewrite: Transform the logical query into an equivalent, more efficient query (lower cost query), e.g.
  - $R \bowtie S$ is the same as $S \bowtie R$
  - $\boldsymbol{\sigma}(R \bowtie S)$ is the same at $\boldsymbol{\sigma}(R) \bowtie \boldsymbol{\sigma}(S)$
  - $\boldsymbol{\delta}(R \bowtie S) = \boldsymbol{\delta}(R) \bowtie \boldsymbol{\delta}(S)$, where $\boldsymbol{\delta}$ is the distinct operator.
- Access path selection: Which index to choose?
- Operator implementation selection:
  - There are several related implementations for each operator
  - For example,
    - *Sort Scan*
    - *Sort Join*
    - *Hash Join*
    - *Hash Distinct*

Query Optimization

physical query plan. While many algorithms for operators have been proposed, they largely fall into three classes:

1. Sorting-based methods (Section 15.4).
2. Hash-based methods (Sections 15.5 and 20.1).
3. Index-based methods (Section 15.6).

In addition, we can divide algorithms for operators into three "degrees" of difficulty and cost:

a) Some methods involve reading the data only once from disk. These are the *one-pass* algorithms, and they are the topic of this section. Usually, they require at least one of the arguments to fit in main memory, although there are exceptions, especially for selection and projection as discussed in Section 15.2.1.

Query Optimization

b) Some methods work for data that is too large to fit in available main memory but not for the largest imaginable data sets. These *two-pass* algorithms are characterized by reading data a first time from disk, processing it in some way, writing all, or almost all, of it to disk, and then reading it a second time for further processing during the second pass. We meet these algorithms in Sections 15.4 and 15.5.

c) Some methods work without a limit on the size of the data. These methods use three or more passes to do their jobs, and are natural, recursive generalizations of the two-pass algorithms. We shall study multipass methods in Section 15.8.

Query Optimization – Query Rewrite and Algorithm Selection

Cannonical Example – SELECT Pushdown

```
def nested_loop_join(self, right_r, on_fields, where_template=None, project_fields=None, optimize=True):

    scan_rows = self.get_row_list()                                  # Table we will scan.
    probe_rows = right_r.get_row_list()                              # Probe for matching rows of current scan row.
    join_result = []

    for l_r in scan_rows:
        on_template = self.__get_on_template__(l_r, on_fields)       # Compute probe query based on current row.
        current_right_rows = right_r.find_by_template(on_template)   # Find rows matching on condition in probe T

        # If the probe returned rows that match on clause for current row
        if current_right_rows is not None and len(current_right_rows) > 0:
            # Merge the l_r row dictionaries into a single dictionary with each right row.
            # The method takes two lists, e.g. cross-product. Wrap the single, current row in a list
            new_rows = self.__join_rows__([l_r], current_right_rows, on_fields)
            join_result.extend(new_rows)

    # We have computed the JOIN based on the ON clause. Now apply the WHERE and PROJECT.
    final_rows = []
    for r in join_result:                                            # Examine every row.
        if self.matches_template(r, where_template):                 # If matches the WHERE template
            r = self.project([r], fields=project_fields)             # PROJECT to get fields
            final_rows.append(r[0])                                  # Add to result tuples.

    # My implementation of operations returns CSVTables. Your implementation does not need to do so.
    join_result = self.__table_from_rows__(
        "JOIN(" + self.__table_name__ + "," + right_r.__table_name__ + ")",
        None,
        final_rows)

    return join_result
```

Baseline:
Nested Loop Join

Cannonical Example – SELECT Pushdown

```
def test_join():
    people_small_tbl = CSVTable.CSVTable("people_small")
    batting_tbl = CSVTable.CSVTable("batting")
    result = people_small_tbl.nested_loop_join(batting_tbl, on_fields=['playerID'],
                    where_template={"nameFirst": "Charlie"})
    print("Result = \n", result)
```

```
Result =
 Name: JOIN(people_small,batting) File: DERIVED
Row count: 5


Sample rows:
birthCity       birthCountry    nameFirst       nameLast        playerID        throws          AB              H               stint           teamID          yearID
Falls City      USA             Charlie         Abbey           abbeych01       L               116             30              1               WAS             1893
Falls City      USA             Charlie         Abbey           abbeych01       L               523             164             1               WAS             1894
Falls City      USA             Charlie         Abbey           abbeych01       L               511             141             1               WAS             1895
Falls City      USA             Charlie         Abbey           abbeych01       L               301             79              1               WAS             1896
Falls City      USA             Charlie         Abbey           abbeych01       L               300             78              1               WAS             1897
```

- Executed on full People.csv
- Requires 1,808 seconds.
- Compares each of 19K rows with each of 104K rows = 2 billion.

Cannonical Example – SELECT Pushdown

```
def test_join_3():
    people_tbl = CSVTable.CSVTable("people")
    batting_tbl = CSVTable.CSVTable("batting")
    start_time = time.time()
    print("Before applying select, number of scan rows = ", len(people_tbl.get_row_list()))
    scan_rows = people_tbl.find_by_template({"nameFirst": "Charlie"})
    people_small_tbl = people_tbl.__table_from_rows__("PeopleSmall", None, scan_rows)
    result = people_small_tbl.join(batting_tbl, on_fields=['playerID'],
                                   where_template={"nameFirst": "Charlie"})
    end_time = time.time()
    print("\nElapsed time to execute join = ", end_time - start_time)
    print("Result = \n", result)
```

```
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6 /Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/Fall2018/W4111-Projects/HW3_new/src/HW3_j
Before applying select, number of scan rows =  19370
Before pushdown, scan table size is =  241
Attempting to pushdown WHERE template =  {"nameFirst": "Charlie"}
After pushdown, scan table size is =  241

Elapsed time to execute join =  16.63890504837036
Result =
 Name: JOIN(PeopleSmall,batting) File: DERIVED
Row count: 925


Sample rows:
birthCity      birthCountry  nameFirst  nameLast  playerID    throws  AB   H    stint  teamID  yearID
Falls City     USA           Charlie    Abbey     abbeych01   L       116  30   1      WAS     1893
Falls City     USA           Charlie    Abbey     abbeych01   L       523  164  1      WAS     1894
Falls City     USA           Charlie    Abbey     abbeych01   L       511  141  1      WAS     1895
Falls City     USA           Charlie    Abbey     abbeych01   L       301  79   1      WAS     1896
Falls City     USA           Charlie    Abbey     abbeych01   L       300  78   1      WAS     1897
...            ...           ...        ...       ...         ...     ...  ...  ...    ...     ...
Carmichael     USA           Charlie    Zink      zinkch01    R       0    0    1      BOS     2008
Canton         USA           Charlie    Ziegler   zieglch01   R       11   3    1      PHI     1900
Canton         USA           Charlie    Ziegler   zieglch01   R       8    2    1      CL4     1899
Philadelphia   USA           Charlie    Young     youngch01   R       9    2    1      BLF     1915
Clinton        USA           Charlie    Wilson    wilsoch02   R       31   10   1      SLN     1935
```

Cannonical Example – SELECT Pushdown

```
def test_join_2():
    people_tbl = CSVTable.CSVTable("people")
    batting_tbl = CSVTable.CSVTable("batting")
    start_time = time.time()
    result = people_tbl.join(batting_tbl, on_fields=['playerID'], where_template={"nameFirst": "Charlie"})
    end_time = time.time()
    print("\nElapsed time to execute join = ", end_time - start_time)
    print("Result = \n", result)
```

```
Before pushdown, scan table size is =  19370
Attempting to pushdown WHERE template =  {"nameFirst": "Charlie"}
After pushdown, scan table size is =  241

Elapsed time to execute join =  16.58244800567627
Result =
 Name: JOIN(people,batting) File: DERIVED
Row count: 925
```

Sample rows:

birthCity	birthCountry	nameFirst	nameLast	playerID	throws	AB	H	stint	teamID	yearID
Falls City	USA	Charlie	Abbey	abbeych01	L	116	30	1	WAS	1893
Falls City	USA	Charlie	Abbey	abbeych01	L	523	164	1	WAS	1894
Falls City	USA	Charlie	Abbey	abbeych01	L	511	141	1	WAS	1895
Falls City	USA	Charlie	Abbey	abbeych01	L	301	79	1	WAS	1896
Falls City	USA	Charlie	Abbey	abbeych01	L	300	78	1	WAS	1897
...	...	...	...	...	...	...	...	...	...	...
Carmichael	USA	Charlie	Zink	zinkch01	R	0	0	1	BOS	2008
Canton	USA	Charlie	Ziegler	zieglch01	R	11	3	1	PHI	1900
Canton	USA	Charlie	Ziegler	zieglch01	R	8	2	1	CL4	1899
Philadelphia	USA	Charlie	Young	youngch01	R	9	2	1	BLF	1915
Clinton	USA	Charlie	Wilson	wilsoch02	R	31	10	1	SLN	1935

- The JOIN detects that pushdown is possible
- Does automatically

Cannonical Example – SELECT Pushdown

- The basic rule: $\boldsymbol{\sigma}(R \bowtie S) = \boldsymbol{\sigma}(R) \bowtie \boldsymbol{\sigma}(S)$
- There are several ways this rewrite can help:
  - Complexity goes from #(R) * #(S) to #($\boldsymbol{\sigma}$(R)) * #($\boldsymbol{\sigma}$(S)) [Note: #(X) is no. of tuples in X]
  - I/O (let B(X) be the number of disk blocks for X):
    - I scan R either way, which means I read EVERY disk block one time, B(R)
    - If I can allocate N buffer frames to the probe table, the probability of a cache hit goes from N/[#(S)] to N/[#($\boldsymbol{\sigma}$(S))], and I may be able to hold the entire $\boldsymbol{\sigma}$(S) in memory.
  - The JOINed table is derived, and does not have indexes. The engine may be able to use indexes for $\boldsymbol{\sigma}$(R) and $\boldsymbol{\sigma}$(S) The basic rule: $\boldsymbol{\sigma}(R \bowtie S) = \boldsymbol{\sigma}(R) \bowtie \boldsymbol{\sigma}(S)$
- The rule $\pi(R \bowtie S) = \pi(R) \bowtie \pi(S)$ may have I/O benefits for probing S. The query execution can cache more of the tuples in buffer frames.
- NOTE: This is an example of query rewrite and *two-pass algorithms*.

Swapping Probe and Scan Tables

```
def define_tables():
    """
    This would be done by a DBA. This code is simulating CREATE TABLE statements.
    The underlying CSV file contains the data. HW3 assumes some other code performs insert, update, delete
    on the data files.
    :return: None
    """
    cleanup()
    cat = CSVCatalog.CSVCatalog()                                    # Connect to catalog to perform DDL

    # Define columns.
    cds = []
    cds.append(CSVCatalog.ColumnDefinition("playerID", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("nameLast", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("nameFirst", column_type="text"))
    cds.append(CSVCatalog.ColumnDefinition("birthCity", "text"))
    cds.append(CSVCatalog.ColumnDefinition("birthCountry", "text"))
    cds.append(CSVCatalog.ColumnDefinition("throws", column_type="text"))

    # CREATE TABLE DDL. Put definition information in our version of INFORMATION_SCHEMA.
    t = cat.create_table("people",
                         "/Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/Fall2018/W4111/Data/People.csv", cds)
    t.define_index("pid_idx", "INDEX", ['playerID'])

    cds = []
    cds.append(CSVCatalog.ColumnDefinition("playerID", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("H", "number", True))
    cds.append(CSVCatalog.ColumnDefinition("AB", column_type="number"))
    cds.append(CSVCatalog.ColumnDefinition("teamID", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("yearID", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("stint", column_type="number", not_null=True))

    t = cat.create_table("batting",
                         "/Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/Fall2018/W4111/Data/Batting.csv", cds)
```

- This index does not help for R ⋈ S if the JOIN column is playerID.
  - I have to look at every row in R.
  - To form probe in S with the current row's playerID.
- But, R ⋈ S is the same as S ⋈ R
- If I
  - Scan S to form the probe query into R on playerID,
  - I can probe via the index.

Swapping Probe and Scan Tables

```
def test_join_4():
    people_tbl = CSVTable.CSVTable("people")
    batting_tbl = CSVTable.CSVTable("batting")
    start_time = time.time()
    result = people_tbl.join(batting_tbl, on_fields=['playerID'])
    end_time = time.time()
    print("\nElapsed time to execute join = ", end_time - start_time)
    print("Result = \n", result)
```

```
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6 /Users/donaldferguson/Dropbo
Swapping scan and probe tables.
Before pushdown, scan table size is =  102816
Attempting to pushdown WHERE template =  null
After pushdown, scan table size is =  102816

Elapsed time to execute join =  1.089634656906128
Result =
 Name: JOIN(people,batting) File: DERIVED
Row count: 102815


Sample rows:
AB          H           playerID    stint       teamID      yearID      bi
4           0           abercda01   1           TRO         1871        Fo
118         32          addybo01    1           RC1         1871        Po
137         40          allisar01   1           CL1         1871        Ph
133         44          allisdo01   1           WS3         1871        Ph
120         39          ansonca01   1           RC1         1871        Ma
...         ...         ...         ...         ...         ...         ..
0           0           zychto01    1           SEA         2016        Mo
164         34          zuninmi01   1           SEA         2016        Ca
523         142         zobribe01   1           CHN         2016        Eu
427         93          zimmery01   1           WAS         2016        Wa
4           1           zimmejo02   1           DET         2016        Au
```

- The JOIN algorithm
  - Detects an index that applies to ON playerID
  - Swaps the scan and probe tables.
- There is no WHERE clause to pushdown.
- But,
  - I have a hash index on people.PlayerID
  - Query cost goes
    - From #(R) * #(S)
    - To #(S) * c, where c is a constant.
    - Would be #(S) * log[#(R)] for B+ Tree
- Also not,
  - Because of the algorithm being a nested loop,
  - The row order changed in the result.

Putting the Pieces Together

```
def test_join_5():
    people_tbl = CSVTable.CSVTable("people")
    batting_tbl = CSVTable.CSVTable("batting")
    start_time = time.time()
    result = people_tbl.join(batting_tbl, on_fields=['playerID'],
        where_template={"nameLast": "Williams", "teamID": "BOS"})
    end_time = time.time()
    print("\nElapsed time to execute join = ", end_time - start_time)
    print("Result = \n", result)
```

- Detects index for ON clause and swaps scan and probe tables.
- Pushes down WHERE clause onto scan table.

```
Swapping scan and probe tables.
Before pushdown, scan table size is =  102816
Attempting to pushdown WHERE template =  {"nameLast": "Williams", "teamID": "BOS"}
After pushdown, scan table size is =  4328

Elapsed time to execute join =  0.0972909927368164
Result =
 Name: JOIN(people,batting) File: DERIVED
Row count: 32


Sample rows:
AB          H           playerID    stint       teamID      yearID      birthCity   birthCountry  nameFirst   nameLast    throws
9           3           willida02   1           BOS         1902        Scranton    USA           Dave        Williams    L
284         68          williri01   1           BOS         1911        Carthage    USA           Rip         Williams    R
85          31          willide01   1           BOS         1924        Portland    USA           Denny       Williams    R
218         50          willide01   1           BOS         1925        Portland    USA           Denny       Williams    R
18          4           willide01   1           BOS         1928        Portland    USA           Denny       Williams    R
...         ...         ...         ...         ...         ...         ...         ...           ...         ...         ...
0           0           willira01   1           BOS         2011        Harlingen   USA           Randy       Williams    L
5           1           willida06   1           BOS         1989        Weirton     USA           Dana        Williams    R
0           0           willist02   1           BOS         1972        Enfield     USA           Stan        Williams    R
69          11          willidi02   1           BOS         1964        St. Louis   USA           Dick        Williams    R
136         35          willidi02   1           BOS         1963        St. Louis   USA           Dick        Williams    R
```

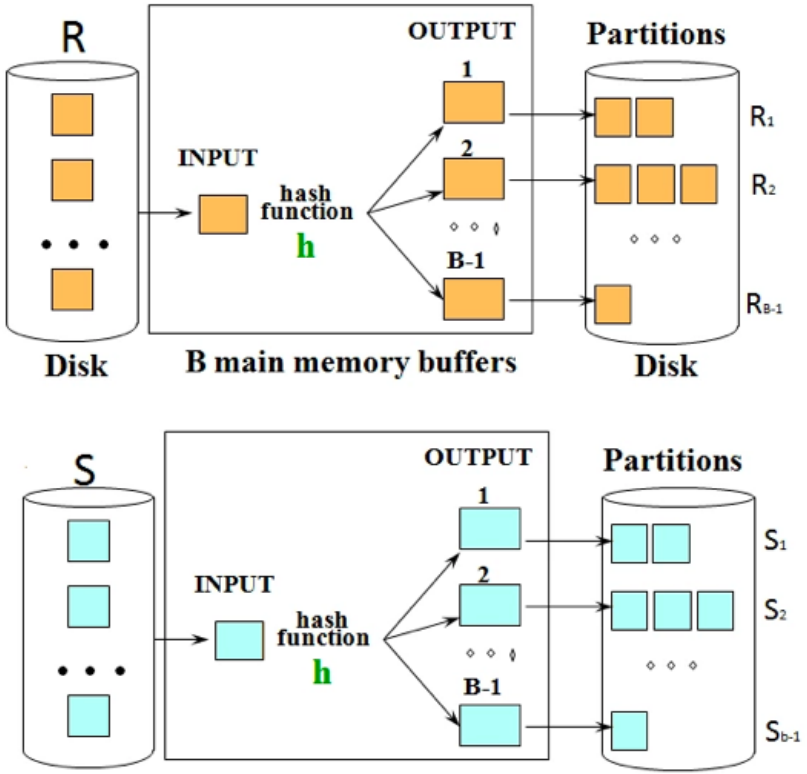
Algorithm Selection

Hash JOIN

- Consider SELECT * FROM people JOIN batting on people.playerID = batting.playerID
- Assume:
  - People has 2 data blocks.
  - Batting has 2 data blocks.
  - I can allocate 3 buffer frames to the JOIN
    - One block holds the current block of R for the scan.
    - One holds a block of S.
    - One holds the JOIN result block I am current constructing.
  - For each row in R, the probability of a buffer hit is 1/2.
  - This means that I will do (1/2) * #(R) I/Os to probe S.
- An alternate approach is to use a hash value on playerID to
  - Read R and partition into 2 buckets, each of size one block.
  - Read S and partition into 2 buckets, each of size one block.
  - I can process the join by loading the 1st block of the hashed R and 1st block of the hashed S into the buffer pool.
  - This means the S tuples matching the current playerID are in the buffer pool.
  - I do some extra upfront I/Os to read, partition and write the hashed tables.
  - But the nested loop is much, much more efficient.
- We still have to scan the "probe" blocks when in memory, but we can hash again.

Hash JOIN (http://cs186.wikia.com/wiki/Joins)

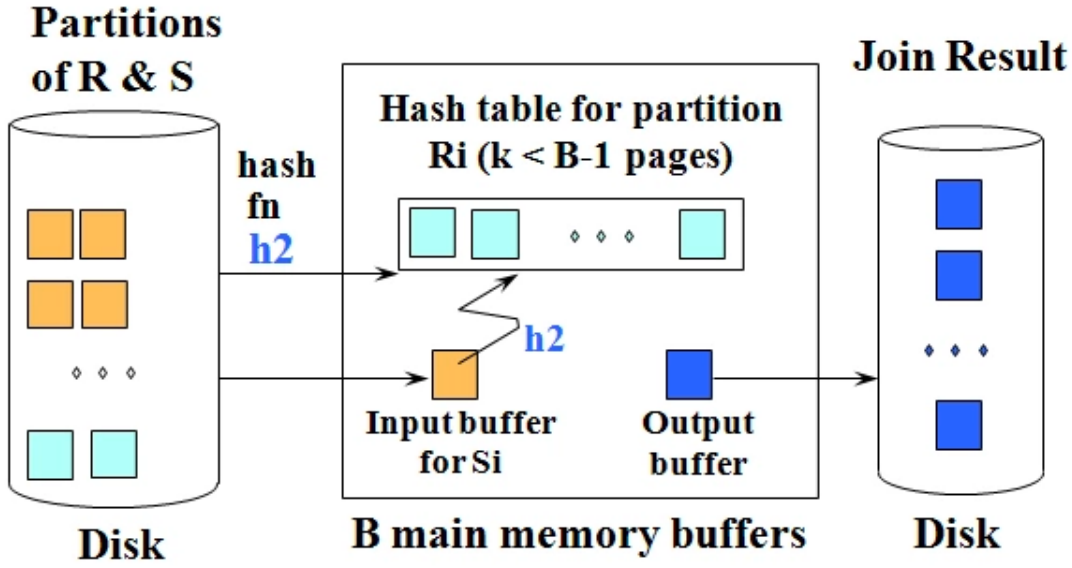
- Hash join uses two stages to accomplish the join.
  - The initial **partitioning** phase,
  - Followed by the **probing** phase.
- In the first stage, we hash the two relations into partitions on disk using a hash function that partitions based on the join condition. The relation *R* gets partitioned into *Ri* and the relation *S* gets partitioned into *Si*. The diagram on the right demonstrates the first stage of the hash join. We will see that in the second stage, we will match *Ri* to *Si* partitions.



Hash JOIN (http://cs186.wikia.com/wiki/Joins)

In the second stage,

- We use an in-memory hash table to perform the joining.
- Read in partition *Ri* and hash it into the in memory hash table
- Scan partition *Si* and probe the in memory hash table for matches.
- Matches go to the output buffer



Summary

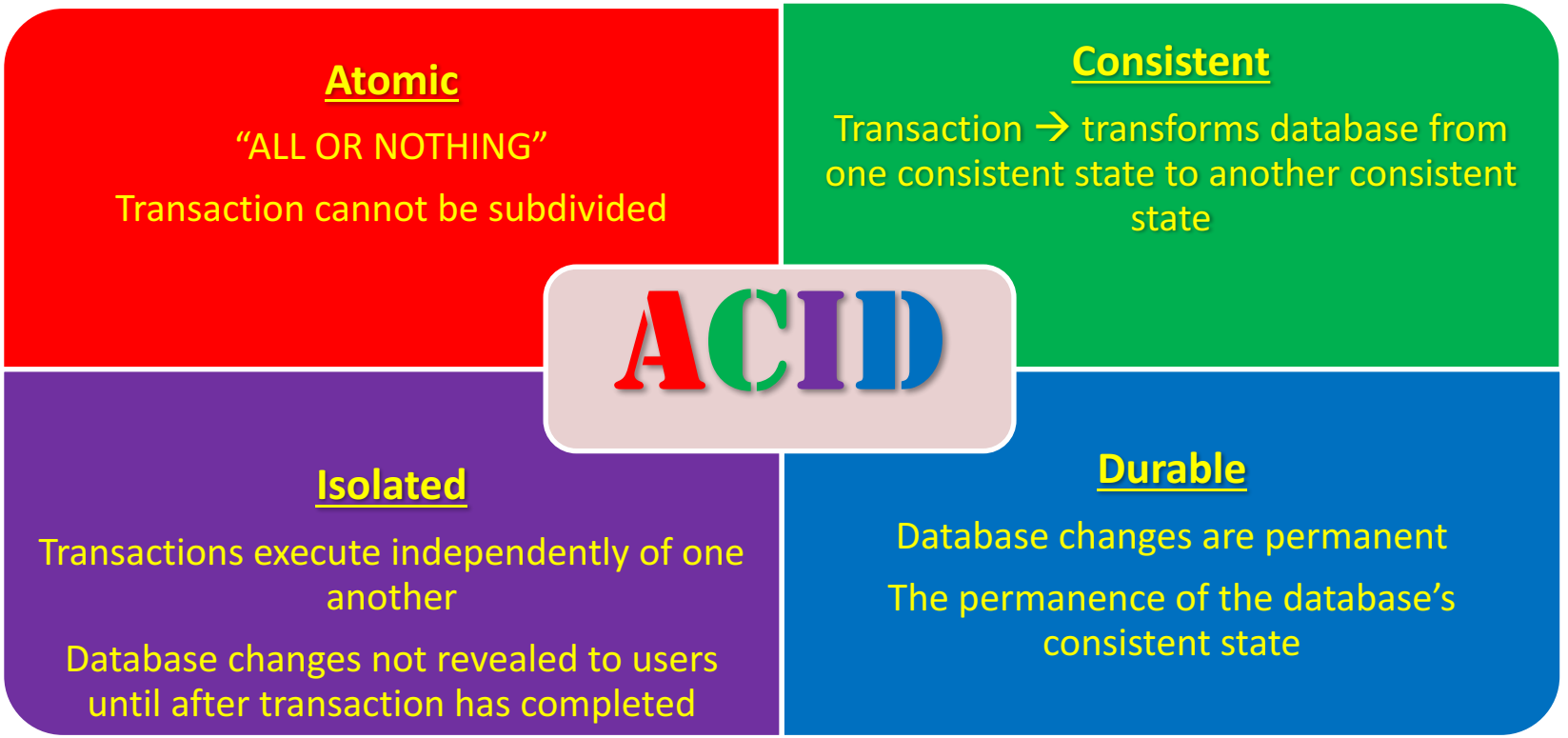
Summary

- Query processing, execution and optimization:
  - Is very powerful.
  - Applies a large body of complex analytics and algorithms to optimize execution.
  - Allows developers to
    - Focus on *declare the result they want using SQL.*
    - Not worry about *how* to efficiently execute the computation.
- Choosing the optimal plan for a query depends on
  - Table sizes.
  - Distribution of column values over ranges.
  - Index selectivity.
  - etc.
- We have just touched the surface but need to move on …

Transactions

Core Transaction Concept is ACID Properties

(http://slideplayer.com/slide/9307681)



ACID Properties (httpwww.tutorialspoint.com/dbms/dbms_transaction.htm)

A *transaction* is a very small unit of a program and it may contain several low-level tasks. A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability – commonly known as ACID properties – in order to ensure accuracy, completeness, and data integrity.

- **Atomicity** – This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency** – The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Durability** – The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- **Isolation** – In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Atomicity

Atomicity

A transaction is a logical unit of work that must be either entirely completed or entirely undone. (All writes happen or none of them happen)

OK. What does this mean? Consider some pseudo-code

def transfer(source_acct_id, target_acct_id, amount)

1. Check that both accounts exist.
2. IF *is_checking_account(source_acct_id)*
    1. Check that (source_acct.balance-amount) > source_account.overdraft_limit
3. ELSE
    1. Check that (source_count.balance-amount) >source_account.minimum_balance
4. Update source account
5. Update target account.
6. INSERT a record into transfer tracking table.

Atomicity

A transaction is a logical unit of work that must be either entirely completed or entirely undone. (All writes happen or none of them happen)

OK. What does this mean? Consider some pseudo-code

def transfer(source_acct_id, target_acct_id, amount)

1. Check that both accounts exist.
2. IF *is_checking_account(source_acct_id)*
    1. Check that (source_acct.balance-amount) > source_acc
3. ELSE
    1. Check that (source_count.balance-amount) >source_ac
4. Update source account
5. Update target account.
6. INSERT a record into transfer tracking table.

Atomicty

- Transaction programs and databases are fast (milliseconds).
  What are the chances of the failure occurring in the wrong spot?
- Well, that doesn't really matter. If it happens,
  - Someone lost money and
  - There is no record off it. Someone is going to very upset.
- Even a small server can have thousands of concurrent transactions ➔
  - There will be corruptions because some transaction will be in the wrong place at the wrong time.
  - Unless we do something in the DBMS
  - Because HW and software inevitably fail
  - And sadly, SW is especially prone to failure when under load

Atomicity

- OK. That stored procedure runs in milliseconds.
  What are the chances of the failure occurring there?
- Well, that doesn't really matter. If it happens,
  - Someone lost money and
  - There is no record off it. Someone is going to very upset.
- Even a small server can have thousands of concurrent transactions ➔
  - There will be corruptions because some transaction will be in the wrong place at the wrong time.
  - Unless we do something in the DBMS
  - Because HW and software inevitably fail
  - And sadly, SW is especially prone to failure when under load

Simplistic Approach

Begin

Update A

Update B

Update C

Commit (REQ)

Force to disk

Commit (RSP)

Simplistic Approach

There are several problems with the simplistic approach.

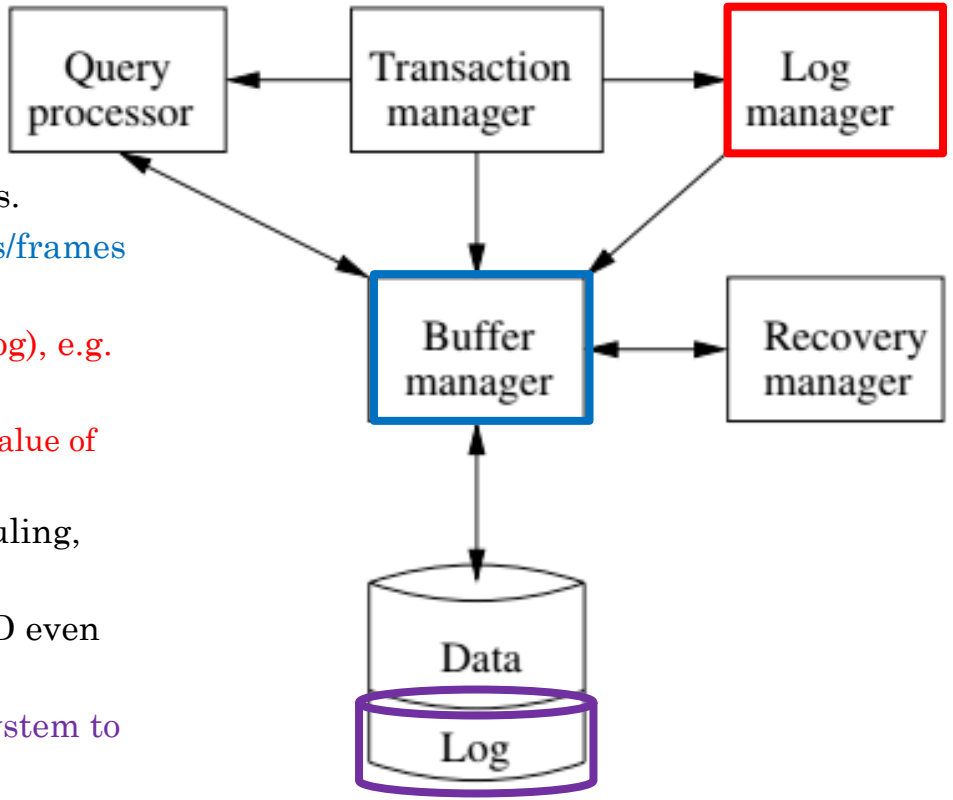
1. The approach does not solve the problem
    1. Some write might succeed.
    2. Some might be interrupted by the failure, or require retry.
2. Writes may be random and scattered. N updates might
    1. Change a few bytes in N data frames
    2. A few bytes in M index frames

    Transaction rate becomes bottlenecked by write I/O rate, even though a relative small number of bytes change/transaction.
3. Written frames must be held in memory.
    1. Lots of transactions
    2. Randomly writing small pieces of lots of frames.
    3. Consumes lots of memory with pinned pages.
    4. Degrades the performance and optimization of the buffer.
        1. The optimal buffer replacement policy wants to hold frames that will be reused.
        2. Not frames that have been touched and never reused.

DBMS ACID Implementation

Implementation Subsystems

- *Query processor* schedules and executes queries.
- *Buffer manager* controls reading/writing blocks/frames to/from disk.
- *Log manager* journals/records events to disk (log), e.g.
  - Transaction start/commit/abort
  - Transaction update of a block and previous value of data.
- *Transaction manager* coordinates query scheduling, buffer read/write and logging to ensure ACID.
- *Recovery manager* processes log to ensure ACID even after transaction or system failures.
- *Log* is a special type of block file used by the system to optimize performance and ensure ACID.



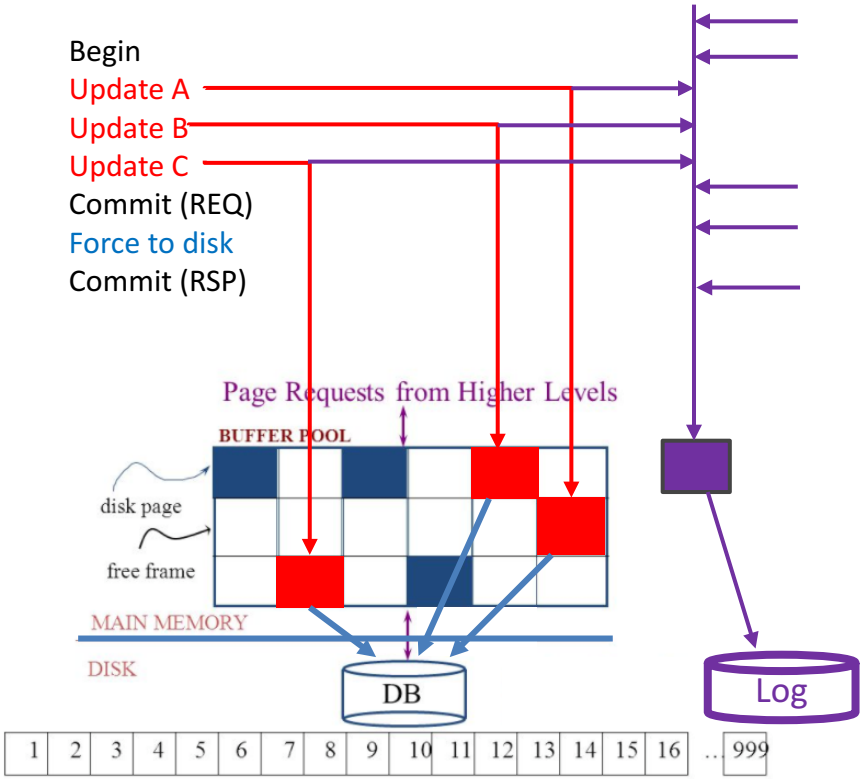
Garcia-Molina et al., p. 846

Logging

The DBMS logs every transaction event

- *Log Sequence Number* (LSN): A unique ID for a log record.
- *Prev LSN*: A link to their last log record.
- *Transaction ID number*.
- *Type*: Describes the type of database log record.
  - **Update Log Record**
    - *PageID*: A reference to the Page ID of the modified page.
    - *Length and Offset*: Length in bytes and offset of the page are usually included.
    - *Before and After Images* of records.
  - **Compensation Log Record**
  - **Commit Record**
  - **Abort Record Checkpoint Record**
  - **Completion Record** notes that all work has been done for this particular transaction.

Write Ahead Logging

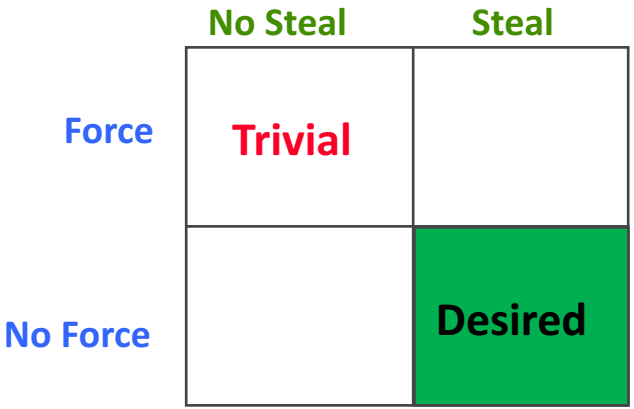


DBMS (Redo processing)

- Write log events from all transactions into a single log stream.
- Multiple events per page
- Forces (writes) log record on COMMIT/ABORT
  - Single block I/O records many updates
  - Versus multiple block I/Os, each recording a single change.
  - All of a transaction's updates recorded in one I/O versus many.
- If there is a failure
  - DBMS sequentially reads log.
  - Applies changes to modified pages that were not saved to disk.
  - Then resumes normal processing.

Write Ahead Logging

- Force every write to disk?
  - Poor response time.
  - But provides durability.
- Steal buffer-pool frames from uncommitted transactions?
  - If not, poor performance/caching performance
  - If yes, how can we ensure atomicity? Uncommitted updates on disk



DBMS (Undo processing)

- Enable steal policy to improve cache performance by
  - Avoiding lots of pinned pages
  - Unlikely to be reused soon.
- Before stealing
  - Force log record to disk.
  - Update log entry has data record
    - Before image
    - After image
- If there is a failure
  - DBMS sequentially reads log.
  - Undoes changes to
    - modified pages, uncommitted pages
    - That were saved to disk.
  - Then resumes normal processing.

ARIES Algorithm =

Algorithms for Recovery and Isolation Exploiting Semantics

ARIES recovery involves three passes

1. Analysis pass:
   - Determine which transactions to undo
   - Determine which pages were dirty (disk version not up to date) at time of
   - RedoLSN: LSN from which redo should start
2. Redo pass:
   - Repeats history, redoing all actions from RedoLSN
     (updated committed but not written changes to pages)
   - RecLSN and PageLSNs are used to avoid redoing actions already reflected on page
3. Undo pass:
   - Rolls back all incomplete transactions (with uncommitted pages written to disk).
   - Transactions whose abort was complete earlier are not undone

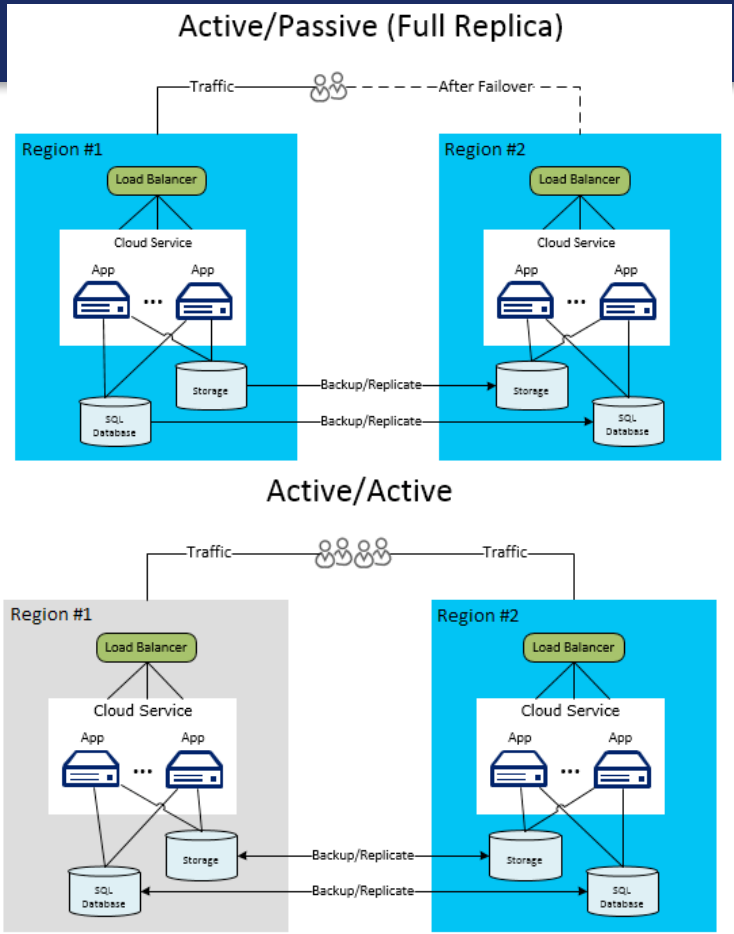
Durability

Durability

- Write changes to disk trivially achieves durability.
- DBMS engine uses write-ahead-logging to
  - Achieve durability
  - But with better performance through more efficient caching and I/O.
- Well, disks fail. How is that durable.
  - RAID and other solutions.
  - Disk subsystems, including entire RAID device, fail →
    - Duplex writes
    - To independent disk subsystems.
- Well, there are earthquakes, floods, etc.

Availability and Replication

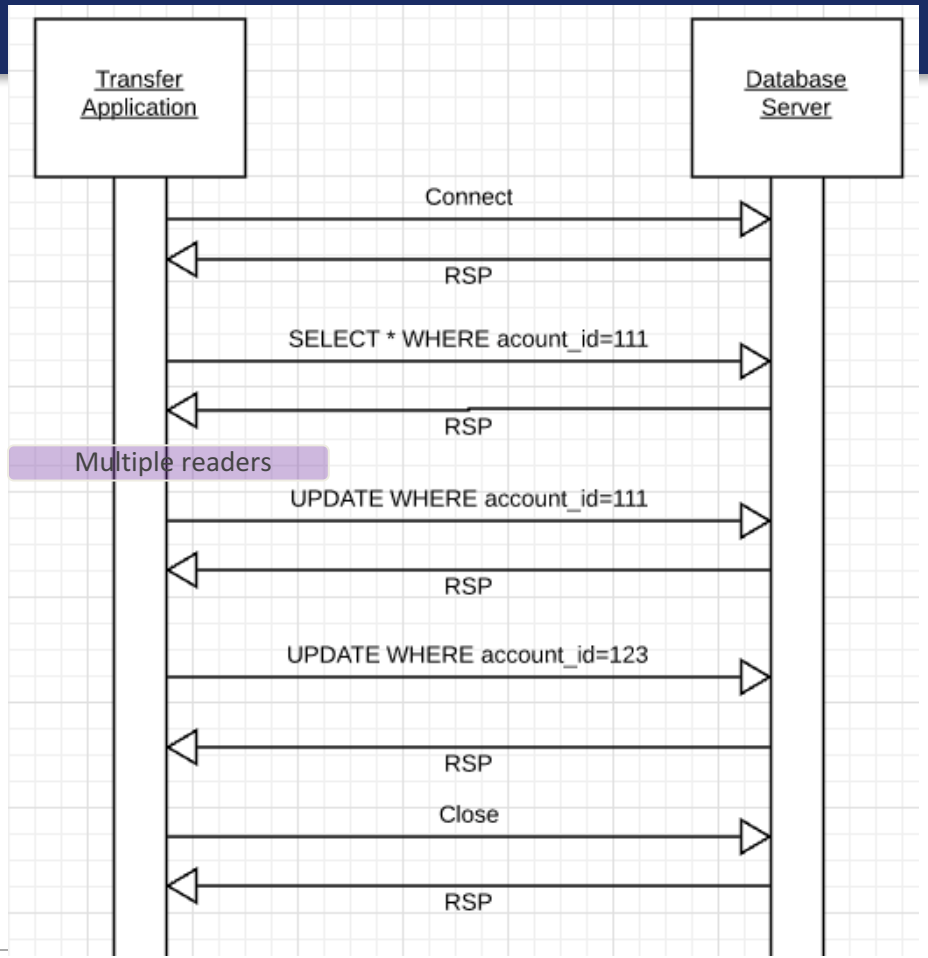
- There are two basic patterns
  - Active/Passive
    - All requests go to *master* during normal processing.
    - Updates are transactionally queued for processing at passive backup.
    - Failure of *master*
      - Routes subsequent requests to *backup*.
      - Backup must process and commit updates before accepting requests.
  - Active/Active
    - Both environments process requests.
    - Some form of distributed transaction commit required to synchronize updates on both copies.
- Multi-system communication to guarantee consistency is the foundation for tradeoffs in CAP.
  - The system can be CAP if and only iff
  - There are never any partitions or system failures
  - Which is unrealistic in cloud/Internet systems.



Isolation

Isolation

- Transfer $50 from
  - account_id=111 to
  - account_id=123
- Requires 3 SQL statements
  - SELECT from 111 to check balance >= $50
  - UPDATE account_id=111
  - UPDATE account_id=123
- There are some interesting scenarios
  - Two different programs read the balance ($51)
  - And decide removing $50 is OK.
- DB constraints can prevent the conflict from happening, but …
  - There are more complex scenarios that constraints do not prevent.
  - Not ALL databases support constraints.
  - The "correct" execution should be that
    - One transaction responds "insufficient funds"
    - Before attempting transfer instead of after attempting.



Isolation

- Consider two **simultaneous** transfer transactions T1 and T2.
- There are two equally ***correct*** executions
  1. T1 transfers, T2 responds "insufficient funds" and does not attempt transfer
  2. T2 transfers, T1 responds "insufficient funds" and does not attempt transfer
- Each correct simultaneous execution is equivalent to a serial (sequential) execution schedule
  - (1) Execute T1, Execute T2
  - (2) Execute T2, Execute T1
- NOTE:
  - We are focusing on ***correctness*** not
  - ***Fairness:***
    - We do not care which transaction was actually submitted first.
    - And probably do not know due to networking, etc.

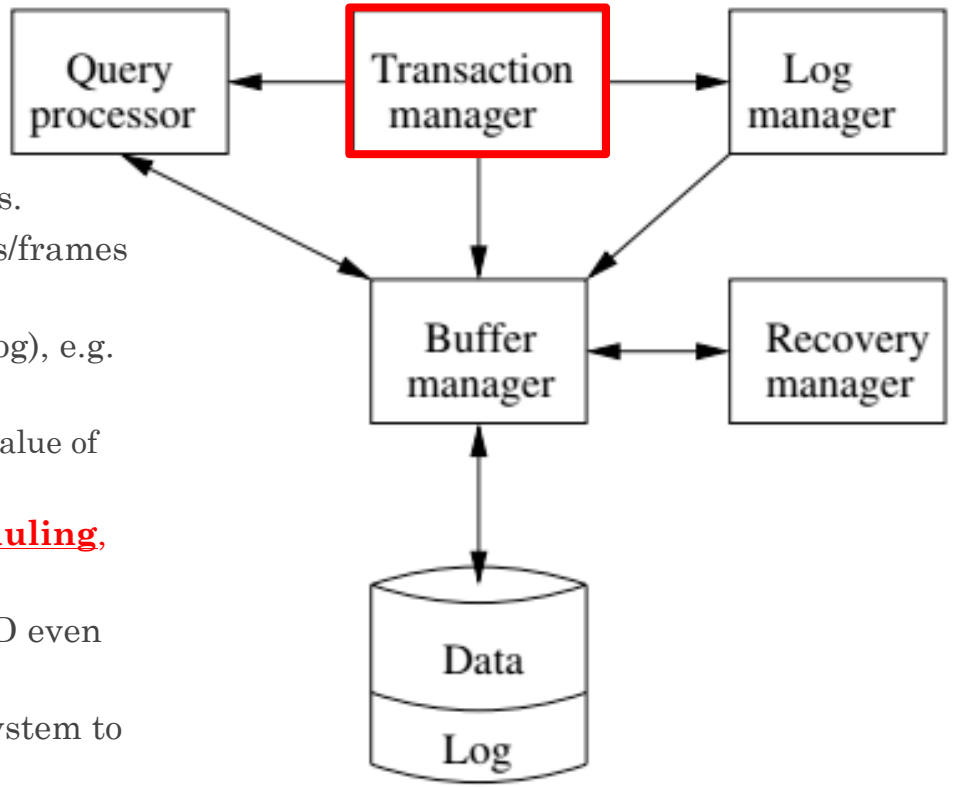
Serializability

"In concurrency control of databases,[1][2] transaction processing (transaction management), and various transactional applications (e.g., transactional memory[3] and software transactional memory), both centralized and distributed, a transaction schedule is **serializable** if its outcome (e.g., the resulting database state) is equal to the outcome of its transactions executed serially, i.e. without overlapping in time. Transactions are normally executed concurrently (they overlap), since this is the most efficient way. Serializability is the major correctness criterion for concurrent transactions' executions. It is considered the highest level of isolation between transactions, and plays an essential role in concurrency control. As such it is supported in all general purpose database systems."
(https://en.wikipedia.org/wiki/Serializability)

DBMS ACID Implementation

Implementation Subsystems

- *Query processor* schedules and executes queries.
- *Buffer manager* controls reading/writing blocks/frames to/from disk.
- *Log manager* journals/records events to disk (log), e.g.
  - Transaction start/commit/abort
  - Transaction update of a block and previous value of data.
- *Transaction manager* coordinates query **scheduling**, buffer read/write and logging to ensure ACID.
- *Recovery manager* processes log to ensure ACID even after transaction or system failures.
- *Log* is a special type of block file used by the system to optimize performance and ensure ACID.



Garcia-Molina et al., p. 846

Schedule

18.1.1 Schedules

A *schedule* is a sequence of the important actions taken by one or more transactions. When studying concurrency control, the important read and write actions take place in the main-memory buffers, not the disk. That is, a database element A that is brought to a buffer by some transaction T may be read or written in that buffer not only by T but by other transactions that access A.

T_1	T_2
READ(A,t)	READ(A,s)
t := t+100	s := s*2
WRITE(A,t)	WRITE(A,s)
READ(B,t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

Figure 18.2: Two transactions

Garcia-Molina et al.

Serializable

- Assume there are three
  - concurrently executing transactions
  - T1, T2 and T3
- The transaction manager
  - Enables concurrent execution
  - But schedules individual operations
  - To ensure that the final DB state
  - Is *equivalent* to one of the following schedules
    - T1, T2, T3
    - T1, T3, T2
    - T2, T1, T3
    - T2, T3, T1
    - T3, T1, T2
    - T3, T2, T1

  Concurrent execution was *serializable*.

18.1.2 Serial Schedules

A schedule is *serial* if its actions consist of all the actions of one transaction, then all the actions of another transaction, and so on. No mixing of the actions is allowed.

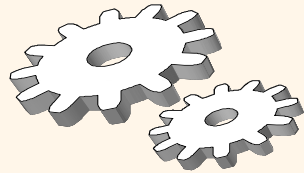
T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

Figure 18.3: Serial schedule in which T_1 precedes T_2

Serializability (en.wikipedia.org/wiki/Serializability)

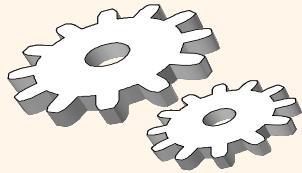
- **Serializability** is used to keep the data in the data item in a consistent state. Serializability is a property of a transaction schedule (history). It relates to the *isolation* property of a database transaction.
- **Serializability** of a schedule means equivalence (in the outcome, the database state, data values) to a *serial schedule* (i.e., sequential with no transaction overlap in time) with the same transactions. It is the major criterion for the correctness of concurrent transactions' schedule, and thus supported in all general purpose database systems.
- **The rationale behind serializability** is the following:
  - If each transaction is correct by itself, i.e., meets certain integrity conditions,
  - then a schedule that comprises any *serial* execution of these transactions is correct (its transactions still meet their conditions):
    - "Serial" means that transactions do not overlap in time and cannot interfere with each other, i.e, complete *isolation* between each other exists.
    - Any order of the transactions is legitimate, (…)
    - As a result, a schedule that comprises any execution (not necessarily serial) that is equivalent (in its outcome) to any serial execution of these transactions, is correct.

Lock-Based Concurrency Control



- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
    - (Non-strict) 2PL Variant: Release locks anytime, but cannot acquire locks after releasing any lock.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.
  - Additionally, it simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing

Aborting a Transaction

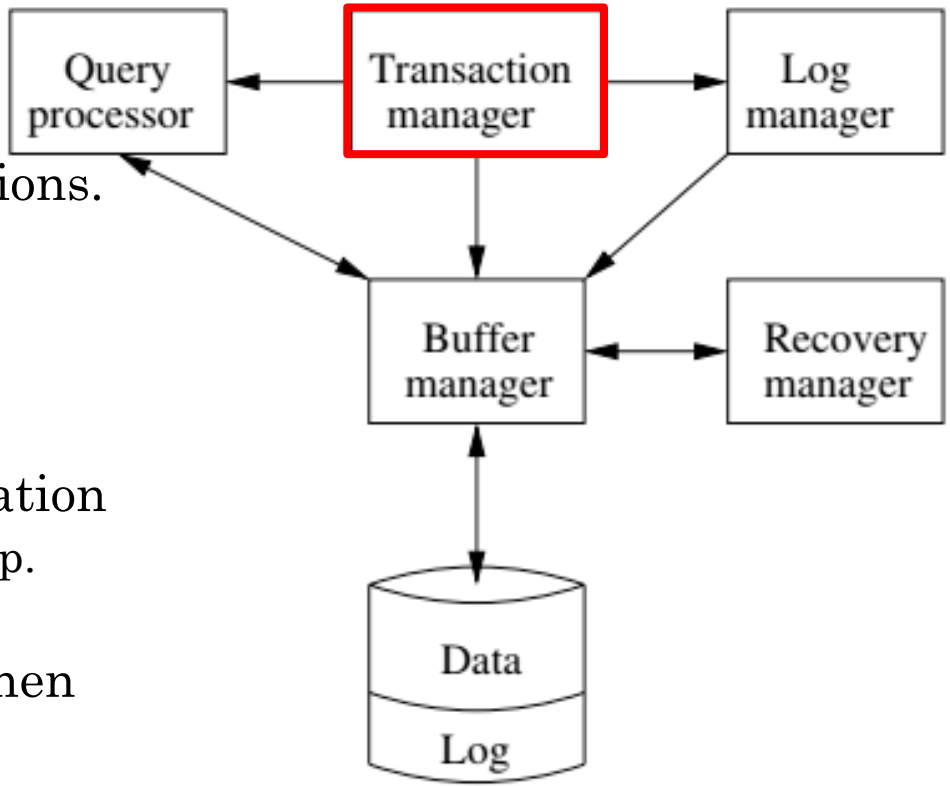


- If a transaction *Ti* is aborted, all its actions have to be undone. Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If *Ti* writes an object, *Tj* can read this only after *Ti* commits.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.

Locking

Transaction Manager

- Intercepts all database operations.
- Acquires/checks locks on
  - Records
  - Pages
  - Index pages
- Suspends and queues an operation
  - In another active, executing op.
  - Has a conflicting lock.
- Restarts queued operations when conflicting locks are released.



García-Molina et al., p. 846

MySQL (Locking) Isolation

13.3.6 SET TRANSACTION Syntax

```
SET [GLOBAL | SESSION] TRANSACTION
    transaction_characteristic [, transaction_characteristic] ...

transaction_characteristic:
    ISOLATION LEVEL level
  | READ WRITE
  | READ ONLY

level:
     REPEATABLE READ
   | READ COMMITTED
   | READ UNCOMMITTED
   | SERIALIZABLE
```

Scope of Transaction Characteristics

You can set transaction characteristics globally, for the current session, or for the next transaction:

- With the `GLOBAL` keyword, the statement applies globally for all subsequent sessions. Existing sessions are unaffected.
- With the `SESSION` keyword, the statement applies to all subsequent transactions performed within the current session.
- Without any `SESSION` or `GLOBAL` keyword, the statement applies to the next (not started) transaction performed within the current session. Subsequent transactions revert to using the `SESSION` isolation level.

Isolation Levels (https://en.wikipedia.org/wiki/Isolation_(database_systems)

Not all transaction use cases require 2PL and serializable execution. Databases support a a set of levels.

- **Serializable**
  - With a lock-based concurrency control DBMS implementation, serializability requires read and write locks (acquired on selected data) to be released at the end of the transaction. Also *range-locks* must be acquired when a SELECT query uses a ranged *WHERE* clause, especially to avoid the ***phantom reads*** phenomenon.
  - *The execution of concurrent SQL-transactions at isolation level SERIALIZABLE is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect as some serial execution of those same SQL-transactions. A serial execution is one in which each SQL-transaction executes to completion before the next SQL-transaction begins.*
- **Repeatable reads**
  - In this isolation level, a lock-based concurrency control DBMS implementation keeps read and write locks (acquired on selected data) until the end of the transaction. However, *range-locks* are not managed, so ***phantom reads*** can occur.
  - Write skew is possible at this isolation level, a phenomenon where two writes are allowed to the same column(s) in a table by two different writers (who have previously read the columns they are updating), resulting in the column having data that is a mix of the two transactions.[3][4]
- **Read committed**
  - In this isolation level, a lock-based concurrency control DBMS implementation keeps write locks (acquired on selected data) until the end of the transaction, but read locks are released as soon as the SELECT operation is performed (so the ***non-repeatable reads phenomenon*** can occur in this isolation level). As in the previous level, *range-locks* are not managed.
  - Putting it in simpler words, read committed is an isolation level that guarantees that any data read is committed at the moment it is read. It simply restricts the reader from seeing any intermediate, uncommitted, 'dirty' read. It makes no promise whatsoever that if the transaction re-issues the read, it will find the same data; data is free to change after it is read.
- **Read uncommitted**
  - This is the *lowest* isolation level. In this level, ***dirty reads*** are allowed, so one transaction may see *not-yet-committed* changes made by other transactions

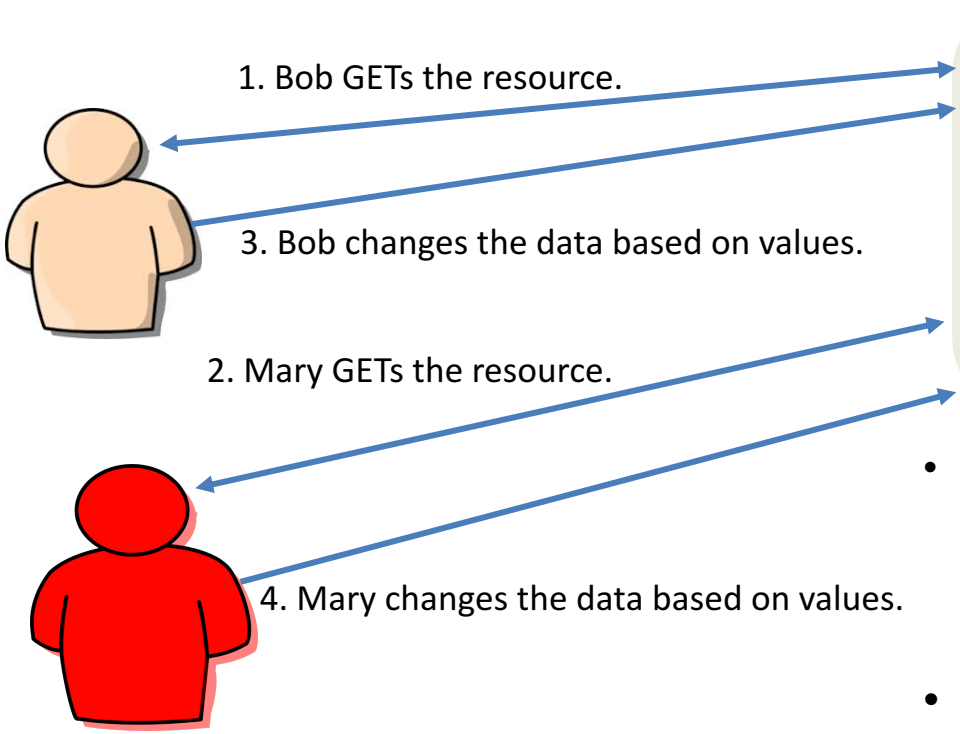
In Databases, Cursors Define *Isolation*

Isolation level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur
Read Committed	-	may occur	may occur
Repeatable Read	-	-	may occur
Serializable	-	-	-

```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource)
ctx.lookup("jdbc/MyBase");
Connection con = ds.getConnection();
DatabaseMetaData dbmd = con.getMetaData();
if (dbmd.supportsTransactionIsolationLevel(TRANSACTION_SERIALIZABLE)
{ Connection.setTransactionIsolation(TRANSACTION_SERIALIZABLE); }
```

- We have talked about ACID transactions
  - Atomic: A set of writes all occur or none occur.
  - Durable: The data "does not disappear," e.g. write to disk.
  - Consistent: My applications move the database between consistent states, e.g. the transfer function works correctly.
- Isolation
  - Determines what happens when two or more threads are manipulating the data at the same time.
  - And is defined relative to where cursors are and what they have touched.
  - Because the cursor movement determines *what you are reading or have read.*
- *But, … Cursors are client conversation state and cannot be used in REST.*

Read then Update Conflict



- If
  - The written value
  - Depends on the read value.
  - The resource may be in the wrong state atthe end.
- The execution was RRWW
- But the intent was RWRW.

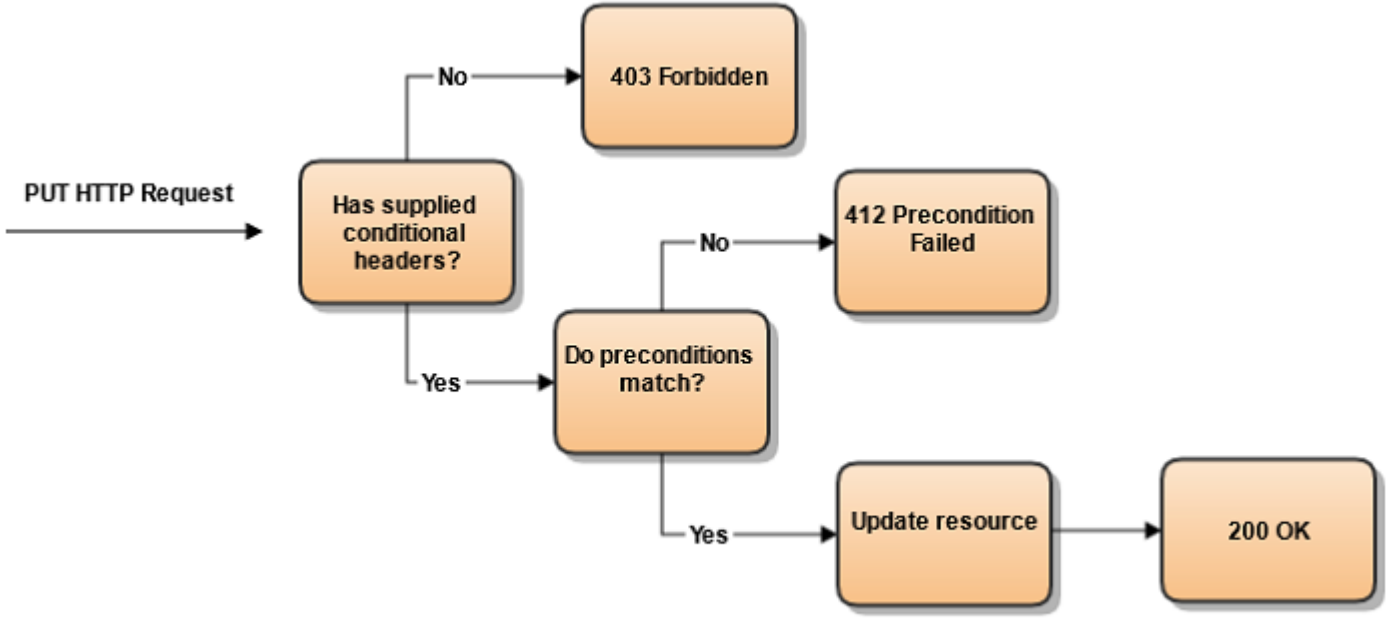
Isolation/Concurrency Control

- There are two basic approaches to implementing isolation
  - Locking/Pessimistic, e.g. cursor isolation
  - Optimistic: Before committing, each transaction verifies that no other transaction has modified the data it has read.
- How does this work in REST?
  - The server maintains an ETag (Entity Tag) for each resource.
  - Every time a resource's state changes, the server computes a new ETag.
  - The server includes the ETag in the header when returning data to the client.
  - The server may *optionally* maintain and return "Last Modified" time for resources.
- Semantics on updates
  - If-Match – value of a previous calls ETag response used with a PUT or DELETE. Server should only act if nobody else has modified the resource since you last fetched it. Otherwise provides a 412.
  - If-Modified-Since – value of a previous Last-Modified response used with a GET. Server should only provide a response if the resource was modified since the timestamp submitted. Use in conjunction with If-None-Match in case the change occurred within the same second. Otherwise provide a 304.
  - If-None-Match – value of a previous calls ETag response used with a GET. Server should only provide a response if the ETag doesn't match, i.e. the resource has been altered. Otherwise provide a 304.
  - If-Unmodified-Since – value of a previous Last-Modified response used with a PUT or DELETE. Server should only act if nobody else has modified the resource since you last fetched it. Otherwise provides a 412

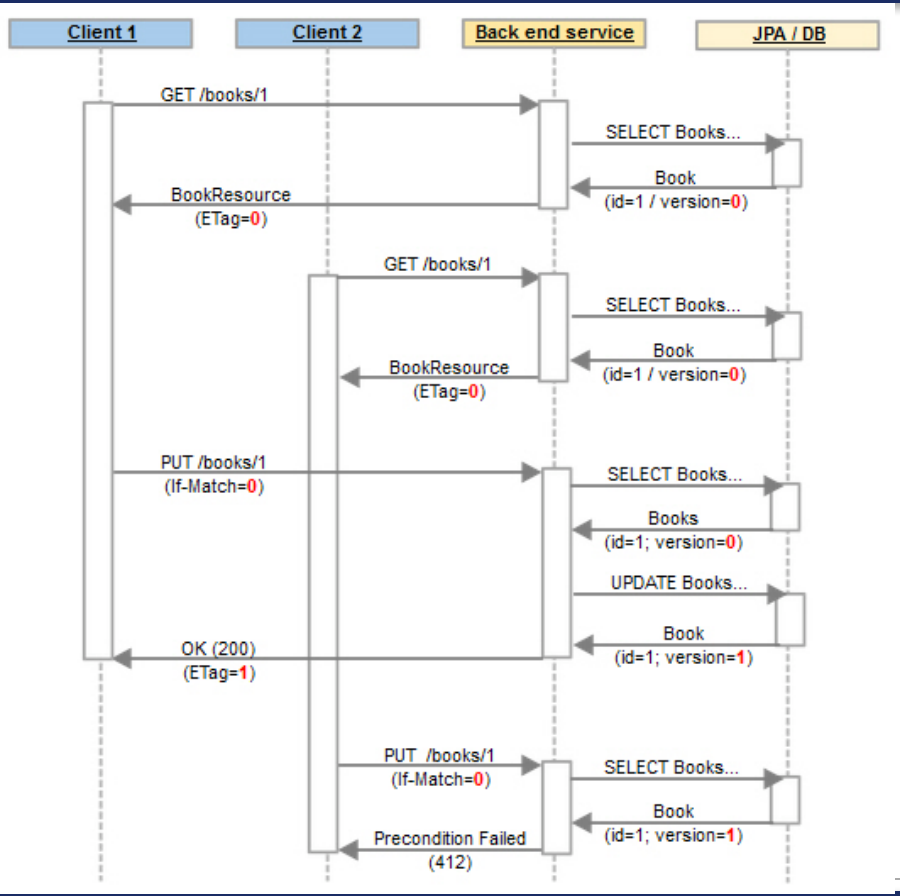
Isolation/Concurrency Control

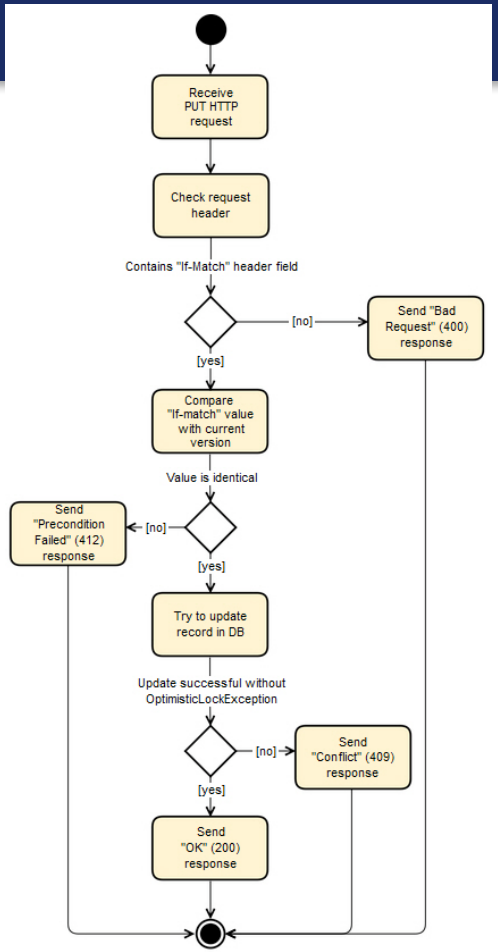
- There are two basic approaches to implementing isolation
  - Locking/Pessimistic, e.g. cursor isolation
  - Optimistic: Before committing, each transaction verifies that no other transaction has modified the data it has read.
- How does this work in REST?
  - The server maintains an ETag (Entity Tag) for each resource.
  - Every time a resource's state changes, the server computes a new ETag.
  - The server includes the ETag in the header when returning data to the client.
  - The server may *optionally* maintain and return "Last Modified" time for resources.
- Semantics on updates
  - If-Match – value of a previous calls ETag response used with a PUT or DELETE. Server should only act if nobody else has modified the resource since you last fetched it. Otherwise provides a 412.
  - If-Modified-Since – value of a previous Last-Modified response used with a GET. Server should only provide a response if the resource was modified since the timestamp submitted. Use in conjunction with If-None-Match in case the change occurred within the same second. Otherwise provide a 304.
  - If-None-Match – value of a previous calls ETag response used with a GET. Server should only provide a response if the ETag doesn't match, i.e. the resource has been altered. Otherwise provide a 304.
  - If-Unmodified-Since – value of a previous Last-Modified response used with a PUT or DELETE. Server should only act if nobody else has modified the resource since you last fetched it. Otherwise provides a 412

Conditional Processing



ETag Processing





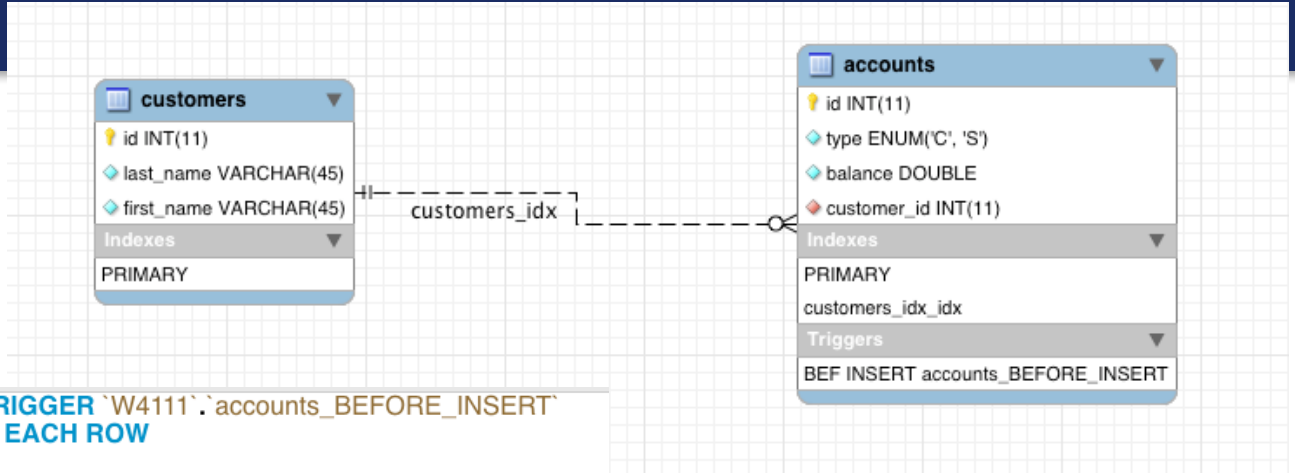
Transaction Examples

MySQL Transaction Operations

13.3.1 START TRANSACTION, COMMIT, and ROLLBACK Syntax

```
START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]

transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY
}

BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET autocommit = {0 | 1}
```

Database




```
CREATE DEFINER=`root`@`localhost` TRIGGER `W4111`.`accounts_BEFORE_INSERT`
    BEFORE INSERT ON `accounts` FOR EACH ROW
BEGIN

    /* Minimum balance on a savings account is $250 */
    if new.type = 'S' and new.balance < 250 then
        signal sqlstate '05001'
            set message_text="Invalid balance.";
    end if;

END
```

```
CREATE DEFINER = CURRENT_USER TRIGGER `W4111`.`accounts_BEFORE_UPDATE`
    BEFORE UPDATE ON `accounts` FOR EACH ROW
BEGIN

    /* Minimum balance on a savings account is $250 */
    if new.type = 'S' and new.balance < 250 then
        signal sqlstate '05001'
            set message_text="Invalid balance.";
    end if;

END
```

Some Code

```python
def get_new_connection():
    cnx = pymysql.connect(
        host=default_dbhost,
        port=default_port,
        user=default_dbuser,
        password=default_dbpw,
        db=default_dbname,
        charset=charset,
        cursorclass=pymysql.cursors.DictCursor)
    return cnx


def commit_cnx(cnx):
    cnx.commit()
    cnx.close()


def abort_cnx(cnx):
    cnx.rollback()
    cnx.close()
```

```python
def insert(cnx, table_name, columns, values, commit=True):
    """
    This is a helper method to perform an insert.
    :param table_name: The RDB table for the insert.
    This is a table in the catalog,
    not one of the CSV table names.
    :param columns: Columns names.
    :param values: Matching values.
    :return: Return value from insert statement
    """
    q = "insert into " + table_name + " "
    column_count = len(columns)
    column_list = ",".join(columns)
    column_list = "(" + column_list + ")"
    v = ["%s"] * column_count
    v = ",".join(v)
    v = " values (" + v + ")"
    q += " " + column_list + " " + v
    rr = run_q(cnx, q, values, False, commit=commit)
    row_id = run_q(cnx,
       "SELECT LAST_INSERT_ID() AS inserted_row_id",
           None, True, commit=commit)
    return row_id
```

Some Code

```
def run_q(cnx, q, args, fetch=False, commit=True):
    """
    :param cnx: The database connection to use.
    :param q: The query string to run.
    :param args: Parameters to insert into query template if q is a template.
    :param fetch: True if this query produces a result and the function shoul
        perform and return fetchall()
    :param commit: If True, commit the transaction
    :return:
    """
    result = None

    try:
        cursor = cnx.cursor()
        print("\nExecuting statement = ", q, "args=", args)
        result = cursor.execute(q, args)
        if fetch:
            result = cursor.fetchall()
        if commit:
            print("\nCommitting for statement = ", q, "args=", args)
            cnx.commit()
    except pymysql_exceptions as original_e:
        raise(original_e)

    return result
```

Create Customer and Account – Failure, No Transaction

```
def create_no_xaction(last_name, first_name, type, balance):
    try:
        print("\nCreating customer and account, no transaction.")
        cnx = dffutils.get_new_connection()
        print("\not connection.")
        r = dffutils.insert(cnx, 'customers', ('last_name', 'first_name'), (last_name, first_name))
        print("\nInserted customer. Inserted row ID = ", r)
        r2 = dffutils.insert(cnx, 'accounts', ('customer_id', 'type', 'balance'),
                             (r[0]['inserted_row_id'], type, balance))
    except Exception as e:
        print("\nException e = ", e)
```

Customer INSERT succeeds but account INSERT fails because savings balance insufficient.

```
def test_xaction_failure():
    print("Testing the failure path with transaction.")
    create_xaction('Bond', 'James', 'S', 100)
    c = get_customer_info('Bond', 'James')
    print("Customer = ", json.dumps(c, indent=2))
```

Create Customer and Account – Failure

```
Testing the failure path.

Creating customer and account, no transaction.

ot connection.

Executing statement =  insert into customers  (last_name,first_name)  values (%s,%s) args= ('Smith', 'John')

Committing for statement =  insert into customers  (last_name,first_name)  values (%s,%s) args= ('Smith', 'John')

Executing statement =  SELECT LAST_INSERT_ID() AS inserted_row_id args= None

Committing for statement =  SELECT LAST_INSERT_ID() AS inserted_row_id args= None

Inserted customer. Inserted row ID =  [{'inserted_row_id': 17}]

Executing statement =  insert into accounts  (customer_id,type,balance)  values (%s,%s,%s) args= (17, 'S', 100)

Exception e =  (1644, 'Invalid balance.')

Executing statement =  select * from customers left join accounts on customers.id=accounts.customer_id  where last_name=%s ar

Committing for statement =  select * from customers left join accounts on customers.id=accounts.customer_id  where last_name=

Customer =  [
  {
    "id": 17,
    "last_name": "Smith",
    "first_name": "John",
    "accounts.id": null,
    "type": null,
    "balance": null,
    "customer_id": null
  }
]
```

Customer INSERT succeeds but account INSERT fails because savings balance insufficient.

Only one of the two intended inserts happened → Not atomic.

Create Customer and Account – Failure, Transaction

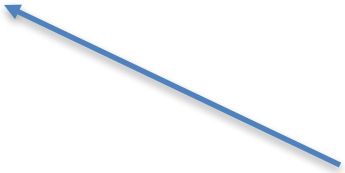
```
def create_xaction(last_name, first_name, type, balance):
    try:
        print("\nCreating customer and account, WITH transaction.")
        cnx = dffutils.get_new_connection()
        print("\ngot connection.")
        r = dffutils.insert(cnx, 'customers', ('last_name', 'first_name'), (last_name, first_name),
                            commit=False)
        print("\nInserted customer. Inserted row ID = ", r)
        r2 = dffutils.insert(cnx, 'accounts', ('customer_id', 'type', 'balance'),
                             (r[0]['inserted_row_id'], type, balance))
        dffutils.commit_cnx(cnx)
    except Exception as e:
        dffutils.abort_cnx(cnx)
        print("Exception e = ", e)
        print("Aborted.")
```

Customer INSERT succeeds but account INSERT fails because savings balance insufficient.
A transaction wraps both inserts.

```
def test_xaction_failure():
    print("\nTesting the failure path with transaction.")
    create_xaction('Bond', 'James', 'S', 100)
    c = get_customer_info('Bond', 'James')
    print("Customer = ", json.dumps(c, indent=2))
```

Create Customer and Account – Failure

```
Creating customer and account, WITH transaction.

got connection.

Executing statement =  insert into customers  (last_name,first_name)  values (%s,%s) args= ('Bond', 'James')

Executing statement =  SELECT LAST_INSERT_ID() AS inserted_row_id args= None

Inserted customer. Inserted row ID =  [{'inserted_row_id': 18}]

Executing statement =  insert into accounts  (customer_id,type,balance)  values (%s,%s,%s) args= (18, 'S', 100)
Exception e =  (1644, 'Invalid balance.')
Aborted.

Executing statement =  select * from customers left join accounts on customers.id=accounts.customer_id  where last_name=%s and first_name=%s args= ('Bond', 'James')

Committing for statement =  select * from customers left join accounts on customers.id=accounts.customer_id  where last_name=%s and first_name=%s args= ('Bond', 'James')
Customer =  []
```



Customer INSERT succeeds but account INSERT fails because savings balance insufficient.

Transaction abort/rollback prevents 1st insert from being permanent.

Isolation would prevent other transactions from seeing un-comitted insert.

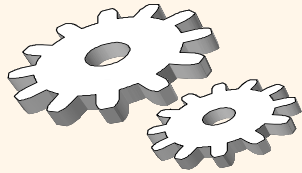
Comments

- In this example, a well designed program would
  - Catch the exception for the failed account insert.
  - Explicitly delete the customer record.
  - To ensure that the transaction moves the DB from one consistent state to another.
- But, there are unexpected conditions that a well designed application cannot easily handle:
  - MySQL fails in between the two inserts.
  - The client system or client application fails in between the two inserts.
  - The network connection breaks in between the two inserts.
  - Another program:
    - Executes the LEFT JOIN in between the two inserts.
    - The program would see an inconsistent state.
- The "A" and "I" in ACID prevent the unexpected condition from causing an inconsistent database state.

Backup

Query execution cost

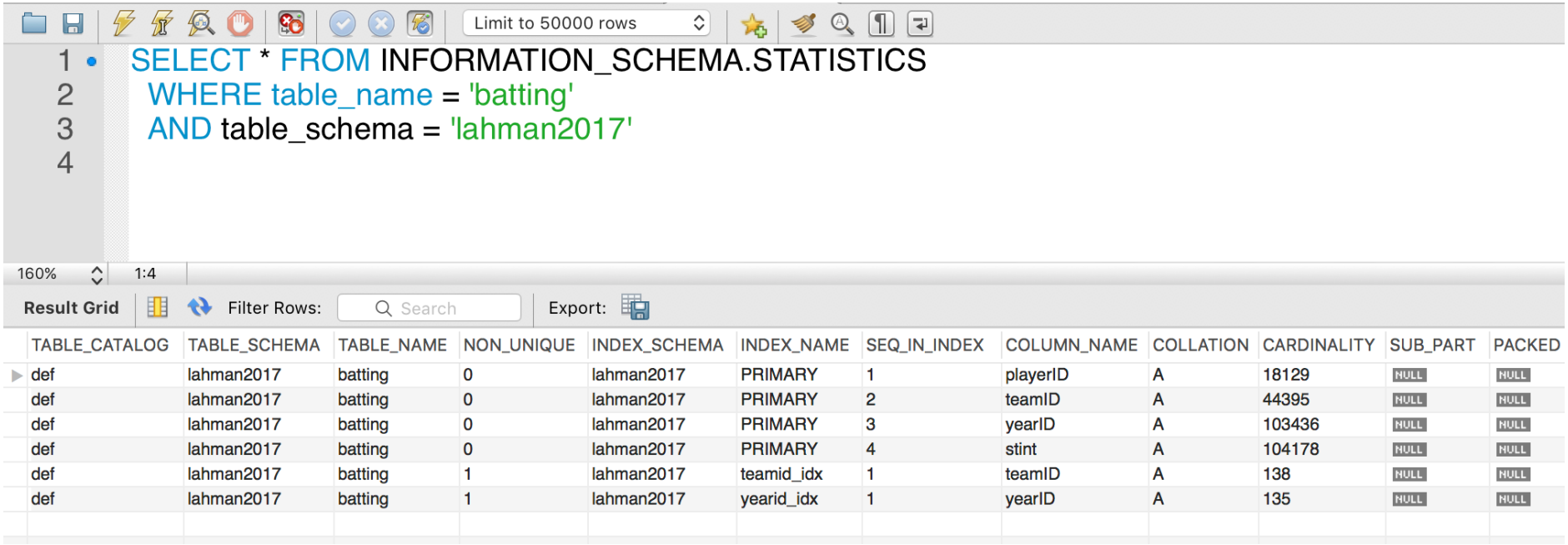
- Query execution cost is usually a weighted sum of the I/O cost (# disk accesses) and CPU cost (msec)
  - w * IO_COST + CPU_COST
- Basic Idea:
  - Cost of an operator depends on input data size, data distribution, physical layout
  - The optimizer uses statistics about the relations to *estimate* the cost
  - Need statistics on base relations and intermediate results



Statistics and Catalogs

- Need information about the relations and indexes involved. ***Catalogs*** typically contain at least:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field) are sometimes stored.

MySQL Example



TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	NON_UNIQUE	INDEX_SCHEMA	INDEX_NAME	SEQ_IN_INDEX	COLUMN_NAME	COLLATION	CARDINALITY	SUB_PART	PACKED
def	lahman2017	batting	0	lahman2017	PRIMARY	1	playerID	A	18129	NULL	NULL
def	lahman2017	batting	0	lahman2017	PRIMARY	2	teamID	A	44395	NULL	NULL
def	lahman2017	batting	0	lahman2017	PRIMARY	3	yearID	A	103436	NULL	NULL
def	lahman2017	batting	0	lahman2017	PRIMARY	4	stint	A	104178	NULL	NULL
def	lahman2017	batting	1	lahman2017	teamid_idx	1	teamID	A	138	NULL	NULL
def	lahman2017	batting	1	lahman2017	yearid_idx	1	yearID	A	135	NULL	NULL

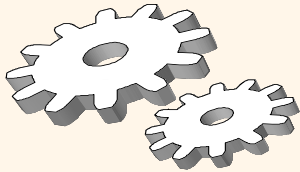
The Catalog Contains

- For each table
  - Table name, storage information.
  - Attribute name and type for each column.
  - Index name, type and indexed attributes
  - Constraints
- Statistical information
  - Cardinality of each table
  - Size: No. of blocks.
  - Index cardinality: Number of distinct key values for each index
  - Index size: Number of blocks for each index.
  - Index tree height
  - Index ranges

Access Path

- Every relational operator accepts one or more tables as input.
  The operator "accesses the tuples" in the tables.
- There are two "ways" to retrieve the tuples
  - Scan the relation (via blocks)
  - Use an index and matching condition.
- A selection (WHERE) condition is in *conjuctive normal form* if
  - Each term is column_name op value
  - op is one of <, <=, =, >, >=, <> (or =, != for Hash Index)
  - The terms are ANDed, e.g. (nameLast = 'Ferguson') AND (ab > 500)
  - CNF allows using a matching index for the access path.
- The engine can select a *matching index*
  - A Hash Index on (c1,c2,c3) if the condition is c1=x AND c2=y AND c3=z.
  - A Tree Index if the condition is (can be ordered as)
    - c1 op value
    - c1 op x AND/OR c2 op y
    - … …
  - Many indexes may match, and the engine chooses the *most selective match* (number of tuples or blocks) that match.

Access Paths



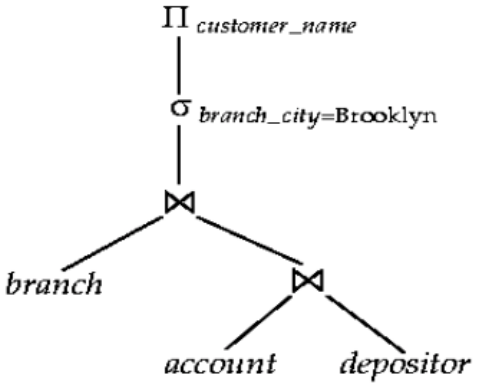
- An access path is a method of retrieving tuples:
  - File scan, or index that matches a selection (in the query)
- A tree index *matches* (a conjunction of) terms that involve only attributes in a *prefix* of the search key.
  - E.g., Tree index on <*a, b, c*> matches the selection *a=5 AND b=3*, and *a=5 AND b>6*, but not *b=3*.
- A hash index *matches* (a conjunction of) terms that has a term *attribute = value* for every attribute in the search key of the index.
  - E.g., Hash index on <*a, b, c*> matches *a=5 AND b=3 AND c=5*; but it does not match *b=3, or a=5 AND b=3, or a>5 AND b=3 AND c=5*.

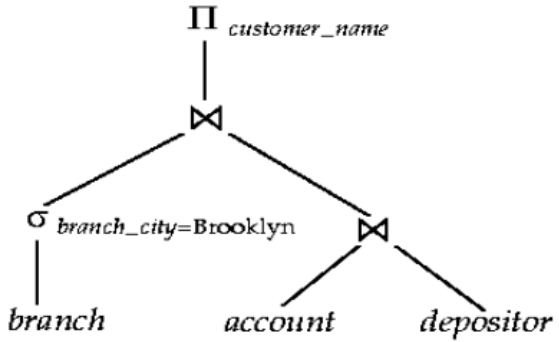
Query Rewrite

Many Ways to Evaluate a Query

Query evaluation

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation





Simple Technique – SELECTion Pushing

- Consider two relations
  - StarsIn(title, year, star_name)
  - Movies(title, year, length, genre, studio_name, producer_no)
- Find everyone who starred in a comedy movie in 1996
  - Query 1:

```
SELECT star_name,  FROM StarsIn JOIN Movies ON
   StarsIn.title=Movies.title AND StarsIn.year=Movies.year
   WHERE StarsIn.year=1996 and Movies.genre='Comedy'
```

  - Query 2:

```
SELECT star_name FROM
   (SELECT star_name, title, year FROM StarsIn WHERE year=1996 as a) JOIN
   (SELECT title, year, genre FROM Movies WHERE year=1996 AND genre='Comedy')
   ON a.year=b.year and a.title=b.title
```

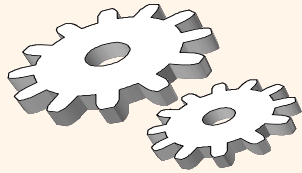
These queries are equivalent, but (2) is much more efficient.

Simple Technique – SELECTion Pushing

- SELECT on a JOIN of table R and table S compares O(#(R)*#(S)) tuples
  - Have to load relevant blocks
  - Compare the tuples to compute the JOIN
  - Scan the JOIN to apply the SELECT
- Pushing the SELECT(s) through the JOINs
  - May vastly reduce the size of the table to JOIN
  - Only JOIN tuples that could be selected in the WHERE clause.

You can also push project through JOIN, but this reduces the size of the data, not the number of tuples examined.

A Note on Complex Selections



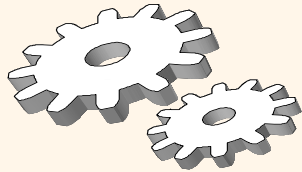
(day<8/9/94 AND rname='Paul') OR bid=5 OR sid=3

- Selection conditions are first converted to *conjunctive normal form* (CNF):

  (day<8/9/94 OR bid=5 OR sid=3) AND
  (rname='Paul' OR bid=5 OR sid=3)

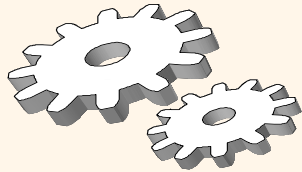
- We only discuss case with no ORs; see text if you are curious about the general case.

One Approach to Selections



- Find the *most selective access path*, retrieve tuples using it, and apply any remaining terms that don't match the index:
  - *Most selective access path:* An index or file scan that we estimate will require the fewest page I/Os.
  - Terms that match this index reduce the number of tuples *retrieved*; other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.
  - Consider *day<8/9/94 AND bid=5 AND sid=3*. A B+ tree index on *day* can be used; then, *bid=5* and *sid=3* must be checked for each retrieved tuple. Similarly, a hash index on <*bid, sid*> could be used; *day<8/9/94* must then be checked.

Using an Index for Selections

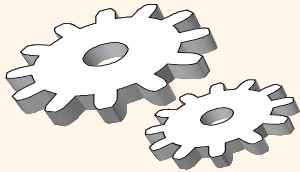


- Cost depends on #qualifying tuples, and clustering.
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering).
  - In example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, upto 10000 I/Os!

```
SELECT  *
FROM    Reserves R
WHERE   R.rname < 'C%'
```

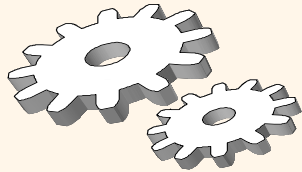
Projection

```
SELECT   DISTINCT
         R.sid, R.bid
FROM     Reserves R
```



- The expensive part is removing duplicates.
  - SQL systems don't remove duplicates unless the keyword DISTINCT is specified in a query.
- Sorting Approach: Sort on <sid, bid> and remove duplicates. (Can optimize this by dropping unwanted information while sorting.)
- Hashing Approach: Hash on <sid, bid> to create partitions. Load partitions into memory one at a time, build in-memory hash structure, and eliminate duplicates.
- If there is an index with both R.sid and R.bid in the search key, may be cheaper to sort data entries!

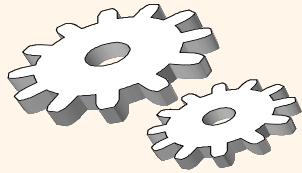
Join: Index Nested Loops



foreach tuple r in R do
    foreach tuple s in S where r_i == s_j do
        add <r, s> to result

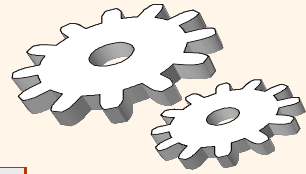
- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: M + ((M*p_R) * cost of finding matching S tuples)
  - M=#pages of R, p_R=# R tuples per page
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple.

Join: Sort-Merge ($R \bowtie_{i=j} S$)



- Sort R and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

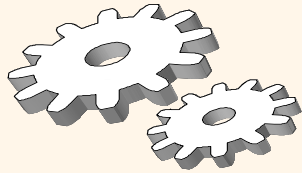
Example of Sort-Merge Join



sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

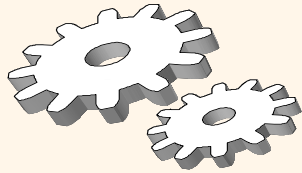
sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- Cost: M log M + N log N + (M+N)
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.



Transactions

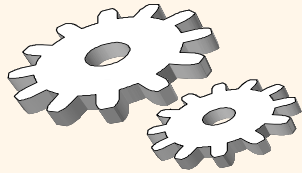
- Concurrent execution of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
- A *transaction* is the DBMS's abstract view of a user program: a sequence of reads and writes.



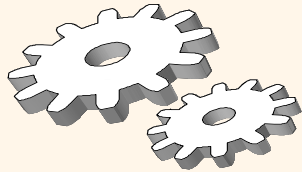
Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- *Issues:* Effect of *interleaving* transactions, and *crashes*.

Atomicity of Transactions



- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- A very important property guaranteed by the DBMS for all transactions is that they are *atomic.* That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
  - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.

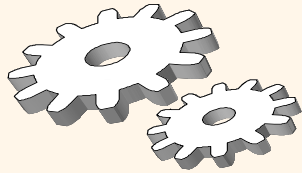


Example

- Consider two transactions (*Xacts*):

```
T1:  BEGIN   A=A+100,   B=B-100   END
T2:  BEGIN   A=1.06*A,   B=1.06*B   END
```

- Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.



Example (Contd.)

- Consider a possible interleaving (*schedule*):

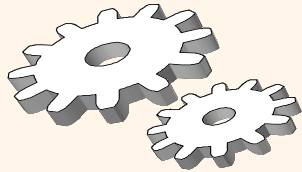
```
T1:   A=A+100,              B=B-100
T2:             A=1.06*A,           B=1.06*B
```

- This is OK. But what about:

```
T1:   A=A+100,                      B=B-100
T2:             A=1.06*A, B=1.06*B
```

- The DBMS's view of the second schedule:

```
T1:   R(A), W(A),                       R(B), W(B)
T2:             R(A), W(A), R(B), W(B)
```

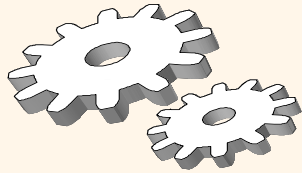


Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules:* For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule:* A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Anomalies with Interleaved Execution

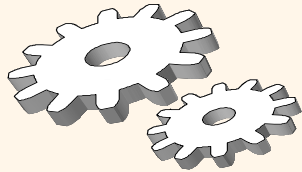


- Reading Uncommitted Data (WR Conflicts, "dirty reads"):

```
T1:  R(A), W(A),                    R(B), W(B), Abort
T2:              R(A), W(A), C
```

- Unrepeatable Reads (RW Conflicts):

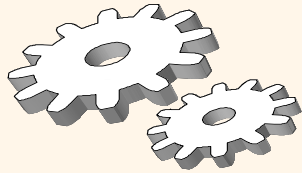
```
T1:  R(A),                  R(A), W(A), C
T2:       R(A), W(A), C
```



Anomalies (Continued)

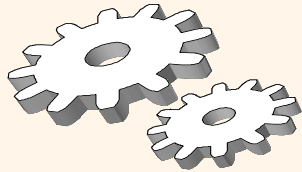
- Overwriting Uncommitted Data (WW Conflicts):

```
T1:  W(A),          W(B), C
T2:       W(A), W(B), C
```



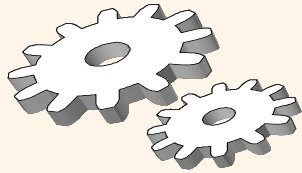
Lock-Based Concurrency Control

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
    - (Non-strict) 2PL Variant: Release locks anytime, but cannot acquire locks after releasing any lock.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.
  - Additionally, it simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing



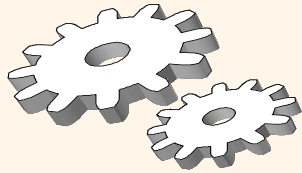
Aborting a Transaction

- If a transaction *Ti* is aborted, all its actions have to be undone. Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If *Ti* writes an object, *Tj* can read this only after *Ti* commits.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.



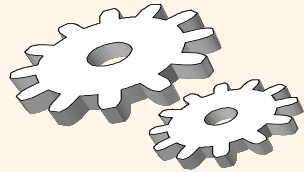
The Log

- The following actions are recorded in the log:
  - *Ti writes an object*: the old value and the new value.
    - Log record must go to disk *before* the changed page!
  - *Ti commits/aborts*: a log record indicating this action.
- Log records are chained together by Xact id, so it's easy to undo a specific Xact.
- Log is often *duplexed* and *archived* on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.



Recovering From a Crash

- There are 3 phases in the *Aries* recovery algorithm:
  - *Analysis*: Scan the log forward (from the most recent *checkpoint*) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
  - *Redo*: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
  - *Undo*: The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)



Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Users need not worry about concurrency.
  - System automatically inserts lock/unlock requests and schedules actions of different Xacts in such a way as to ensure that the resulting execution is equivalent to executing the Xacts one after the other in some order.
- Write-ahead logging (WAL) is used to undo the actions of aborted transactions and to restore the system to a consistent state after a crash.
  - *Consistent state*: Only the effects of commited Xacts seen.