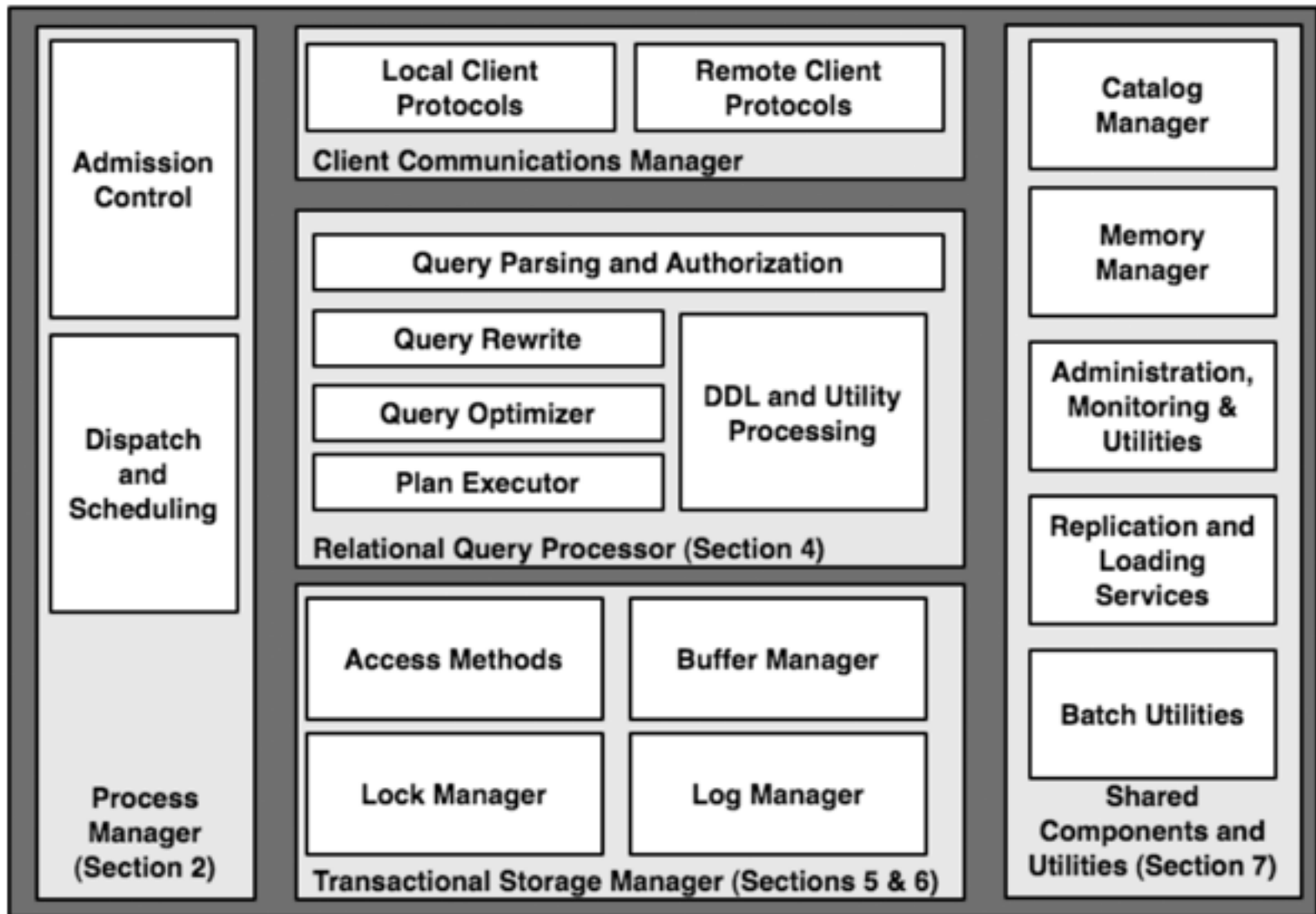


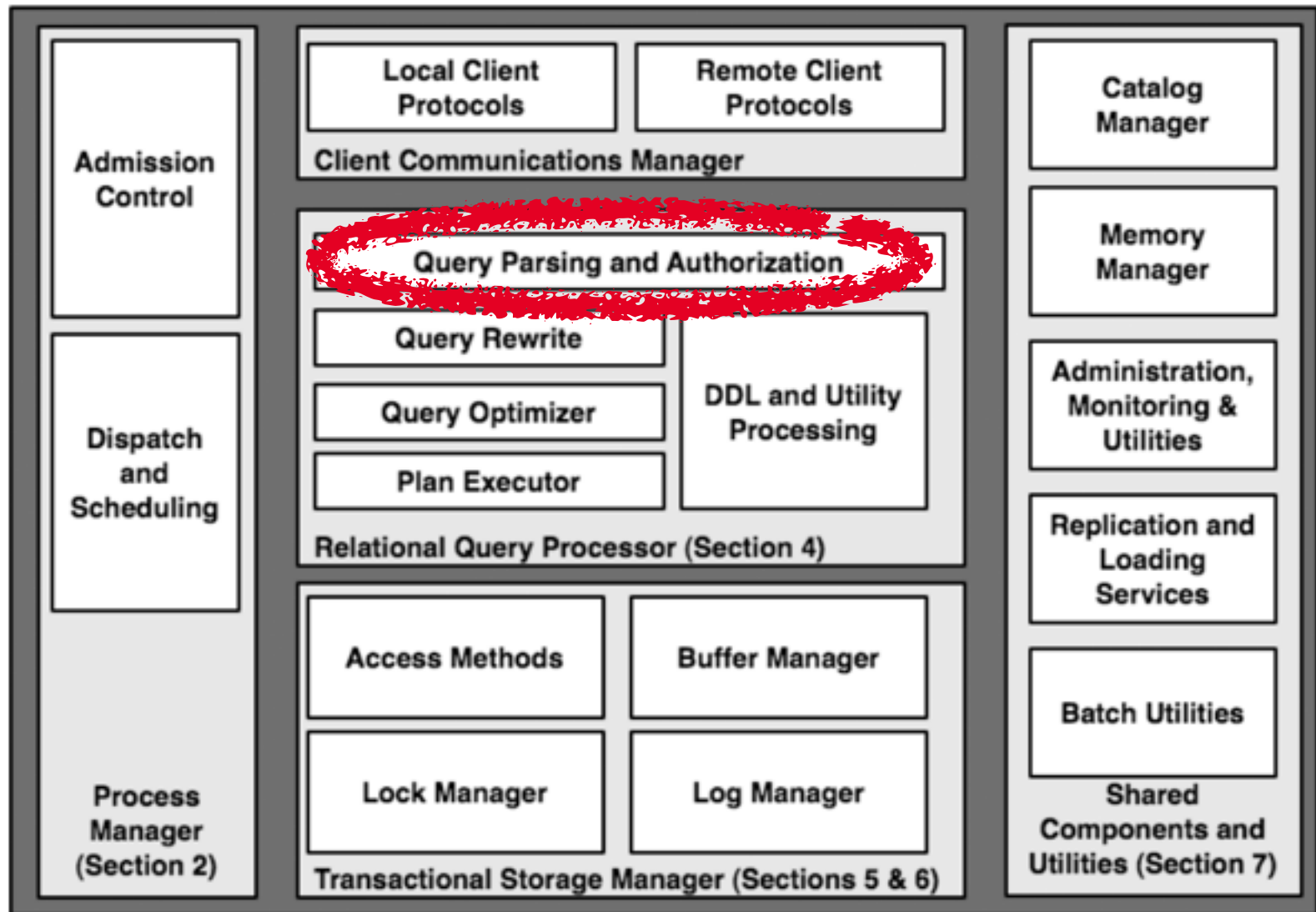
*Overview of Relational DBMS*  
*(CS 4320 Recap)*

CS 6320

# Overview



# Overview





# *Creating Relations in SQL*

- ▶ **Creates Students relation**
  - Type (domain) of each field is specified
  - Enforced by DBMS whenever tuples are added or modified
- ▶ **Enrolled table holds information about courses that students take**

```
CREATE TABLE Students  
  (sid CHAR(20),  
   name CHAR(20),  
   login CHAR(10),  
   age INT,  
   gpa REAL);
```

```
CREATE TABLE Enrolled  
  (sid CHAR(20),  
   cid CHAR(20),  
   grade CHAR(2));
```

# Foreign Keys in SQL

☒ Only students listed in the Students relation should be allowed to enroll for courses

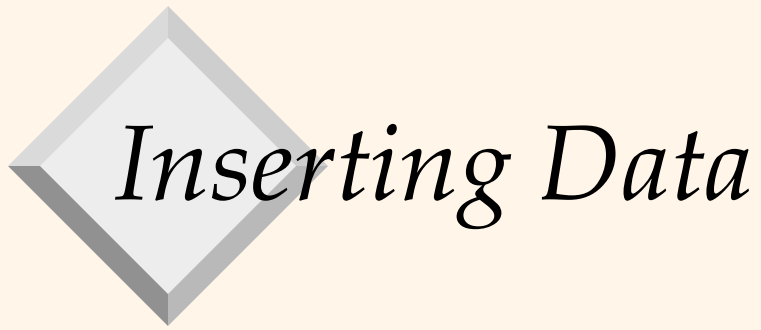
```
CREATE TABLE Enrolled
  (sid CHAR(20), cid CHAR(20), grade CHAR(2),
   PRIMARY KEY (sid,cid),
   FOREIGN KEY (sid) REFERENCES Students (sid) );
```

## Enrolled

sid	cid	grade
53666	Carnatic101	C
53666	Reggae203	B
53650	Topology112	A
53666	History105	B


## Students

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8



# *Inserting Data*

```
INSERT INTO Students  
VALUES ('5', 'Thomas', 'Th75', 20, 3.7);
```



# Querying Data

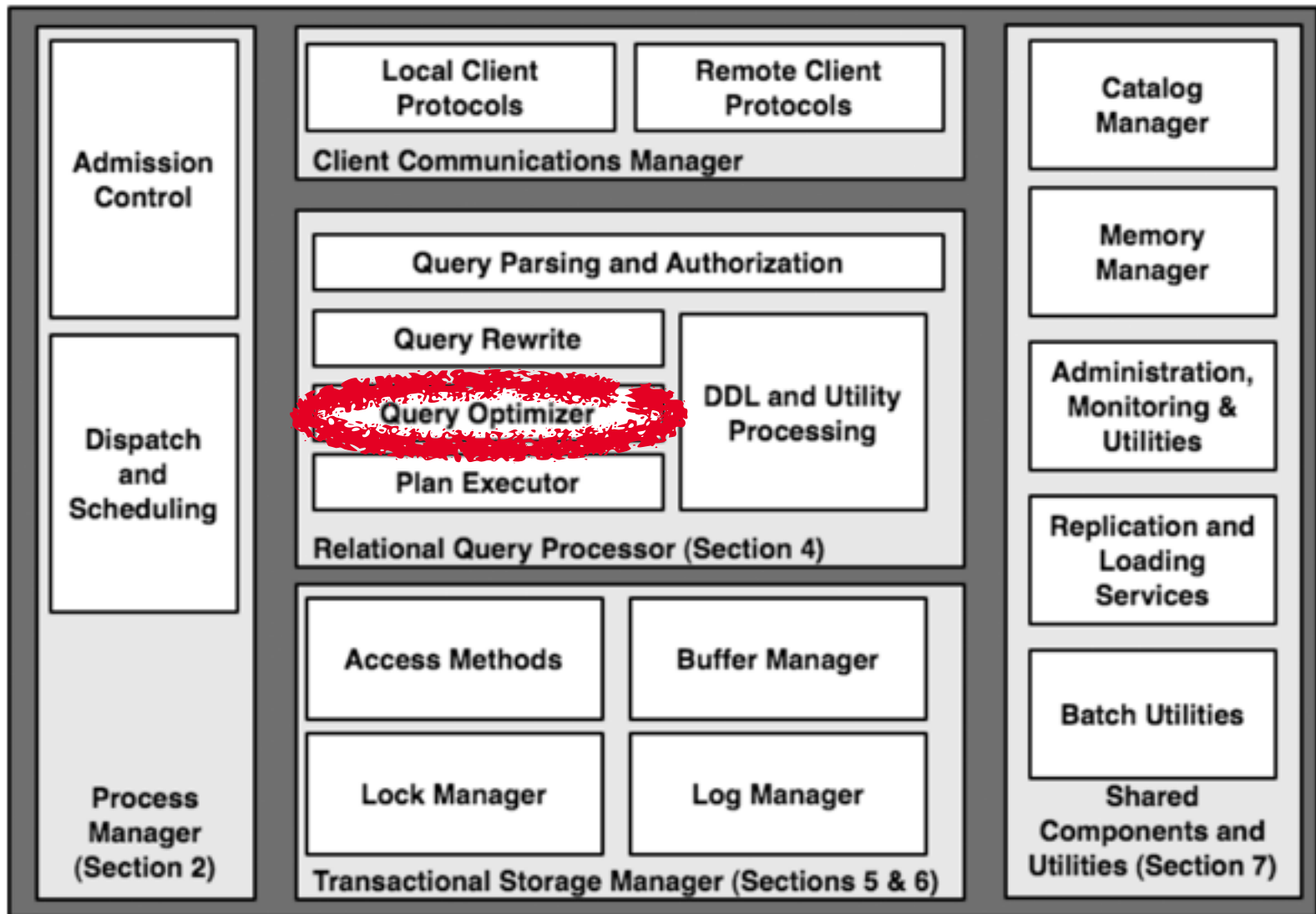
```
SELECT S.name, E.cid  
FROM   Students S, Enrolled E  
WHERE  S.sid=E.sid AND S.gpa>3.5;
```

# *SQL Summary:*

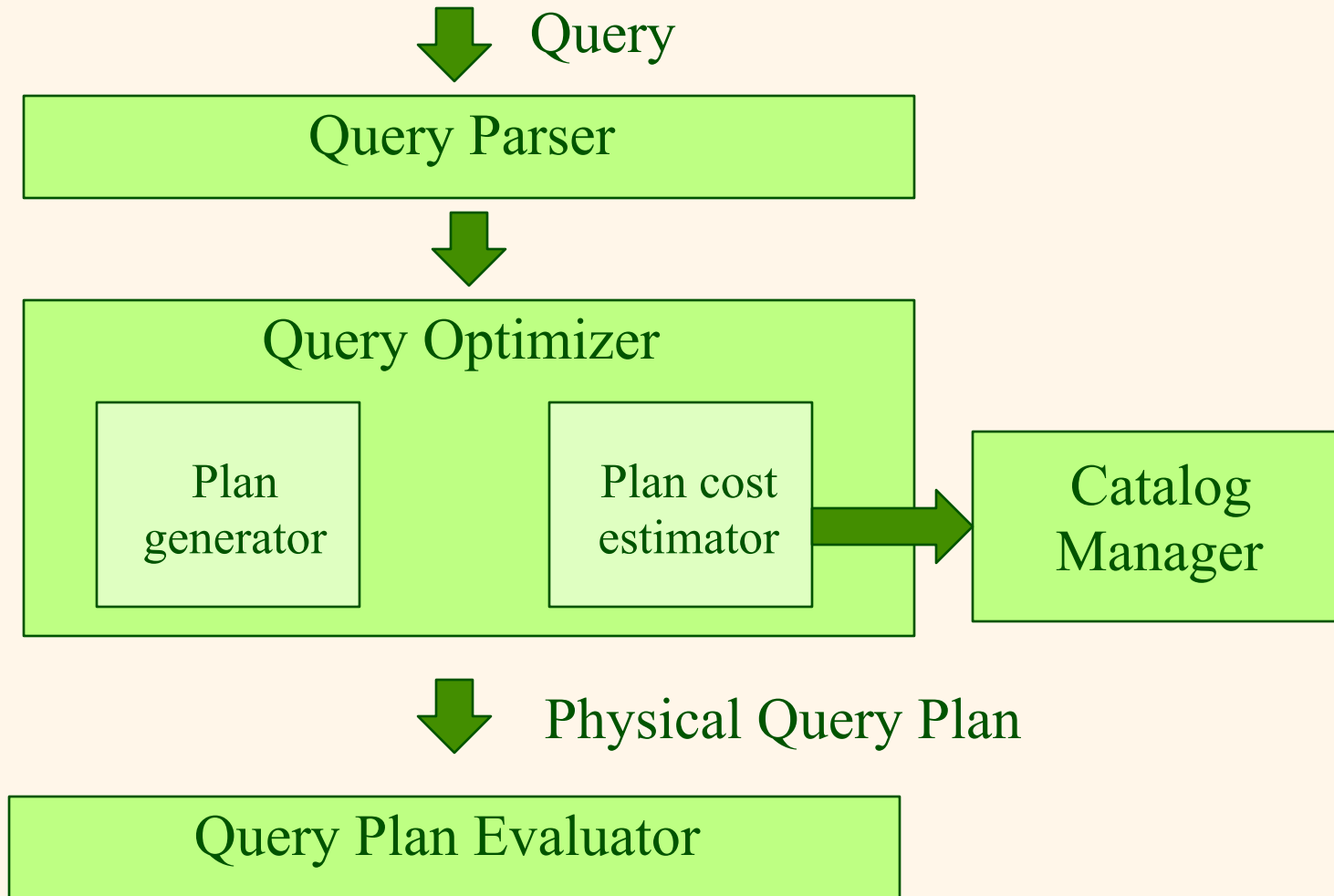
- ❖ Basic SELECT/FROM/WHERE queries
- ❖ Expressions and strings
- ❖ Set operators
- ❖ Nested queries
- ❖ Aggregation
- ❖ GROUP BY/HAVING
- ❖ Null values and Outer Joins
- ❖ (ORDER BY and other features...)



# Overview

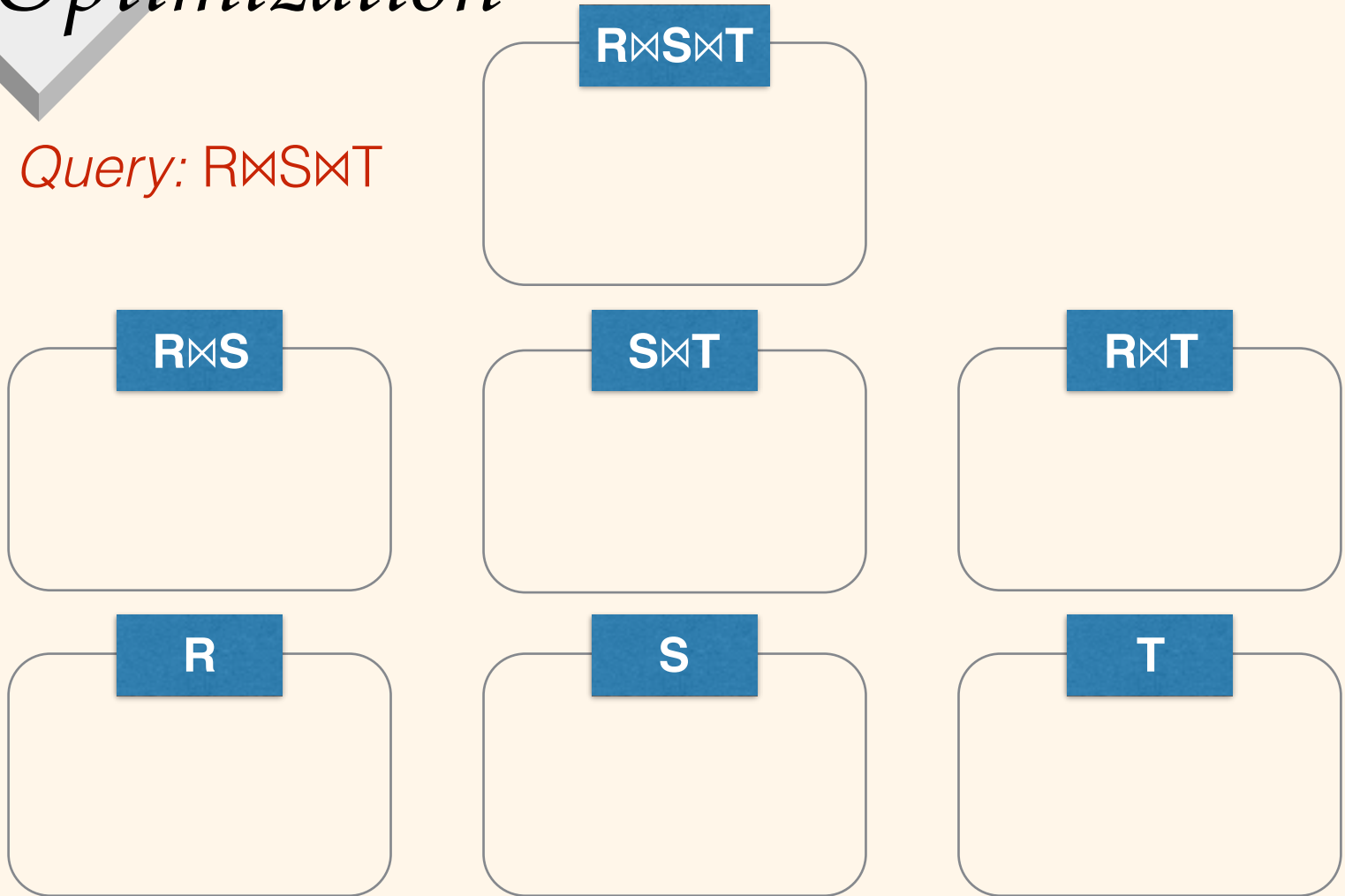


# Query Optimization Overview



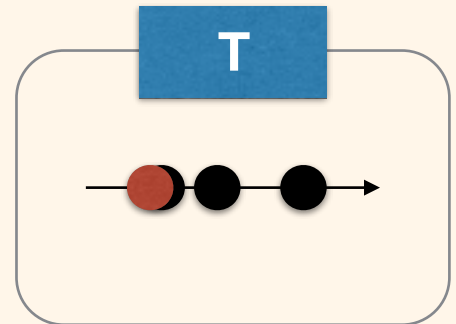
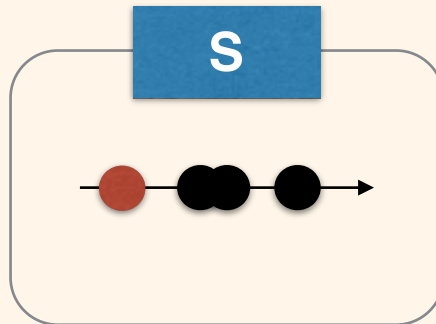
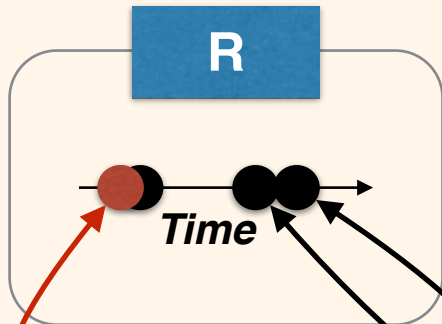
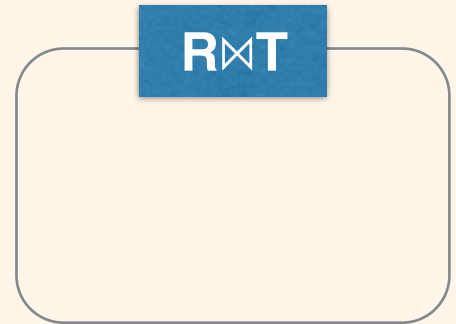
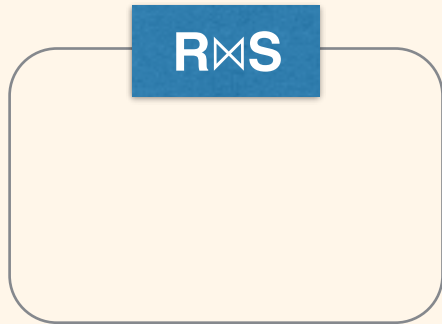
# Optimization

Query:  $R \bowtie S \bowtie T$



# Optimization

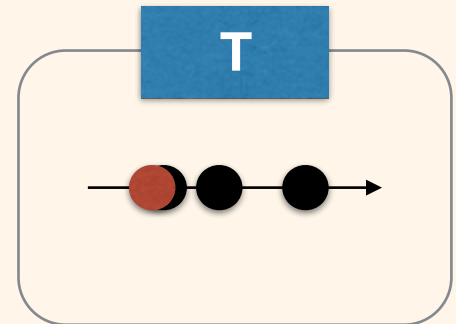
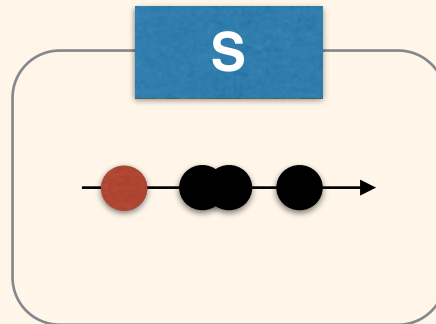
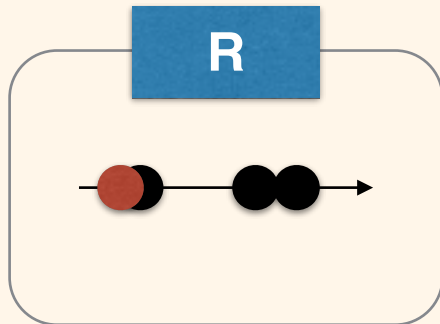
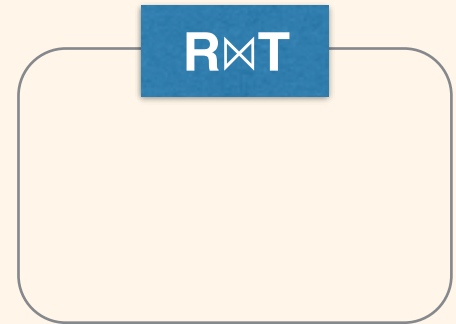
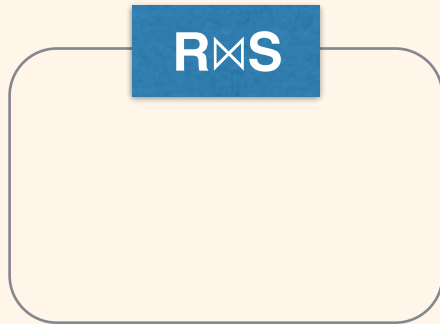
Query:  $R \bowtie S \bowtie T$



**Optimal Plan**      **Sub-Optimal Plans**

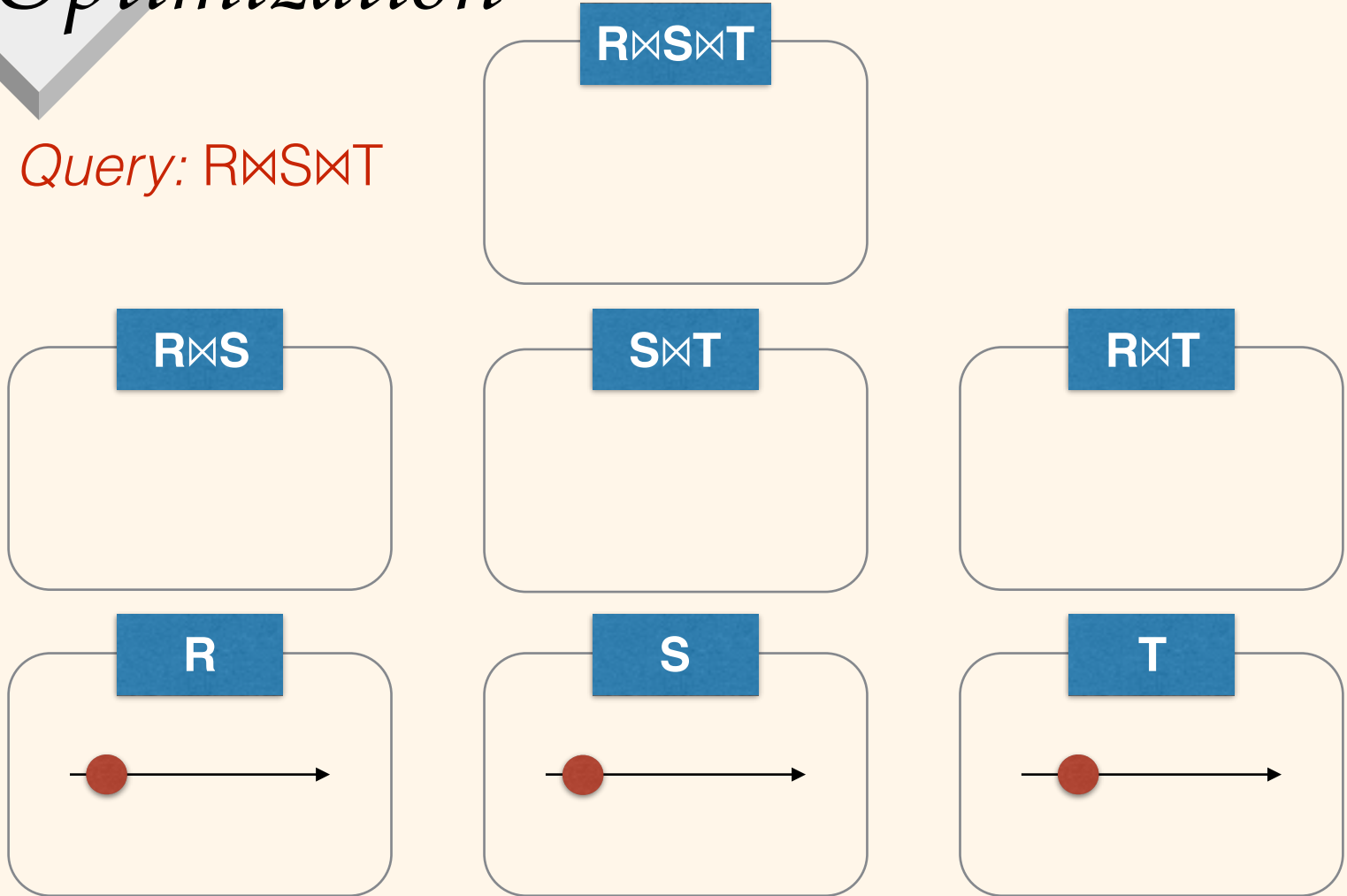
# Optimization

Query:  $R \bowtie S \bowtie T$



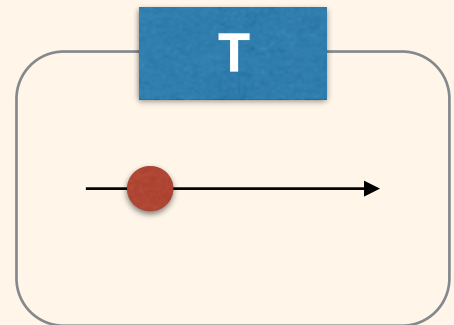
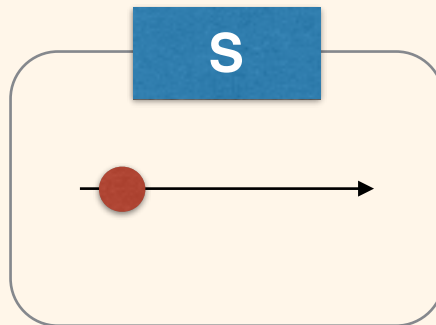
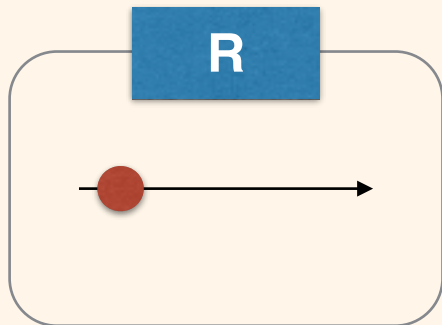
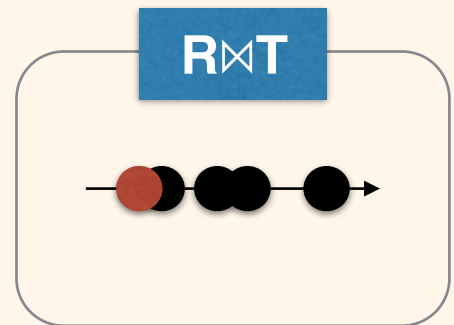
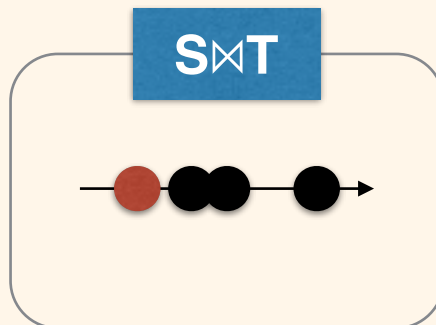
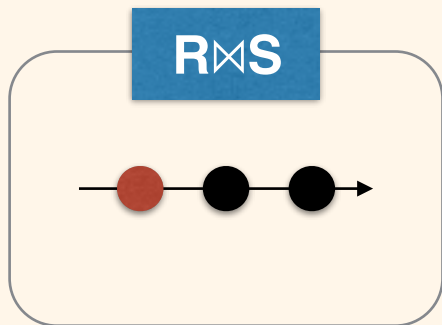
# Optimization

Query:  $R \bowtie S \bowtie T$



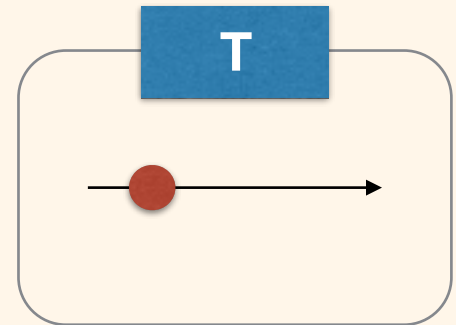
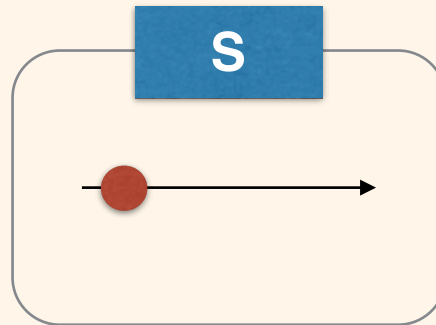
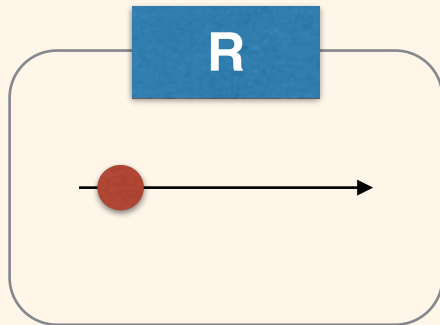
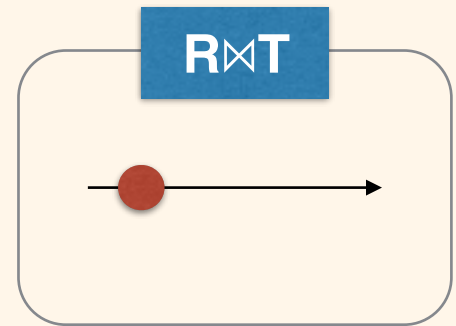
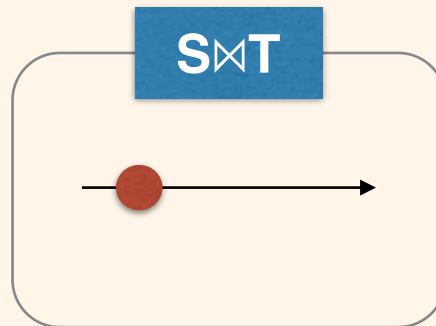
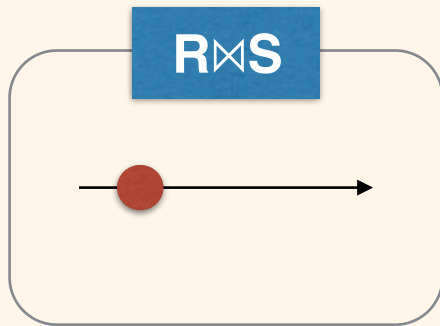
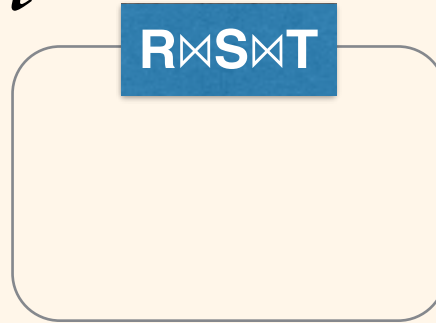
# Optimization

Query:  $R \bowtie S \bowtie T$



# Optimization

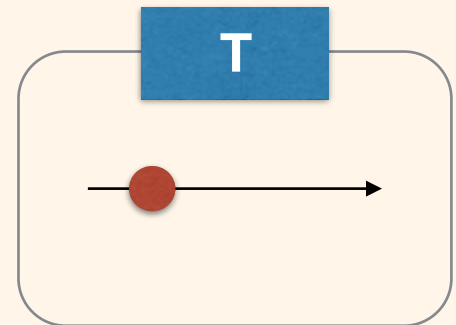
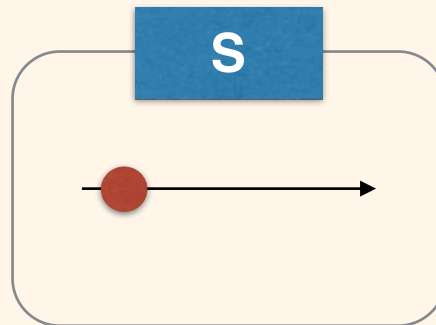
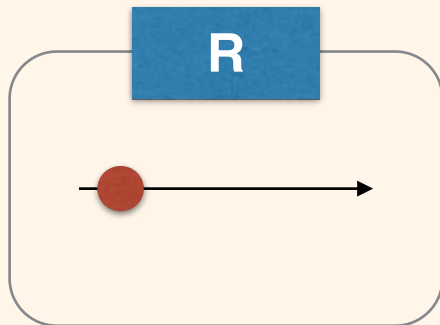
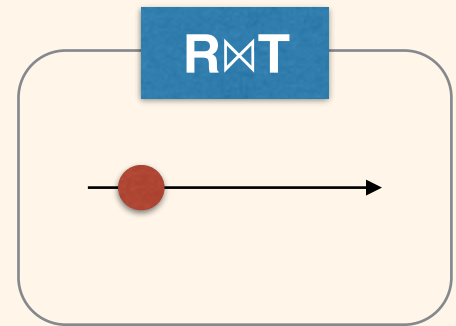
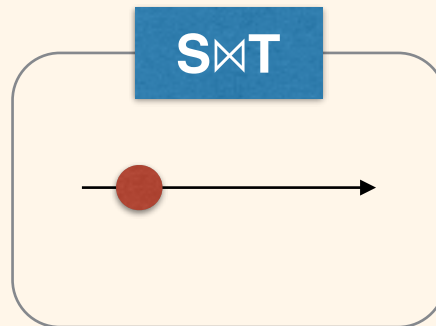
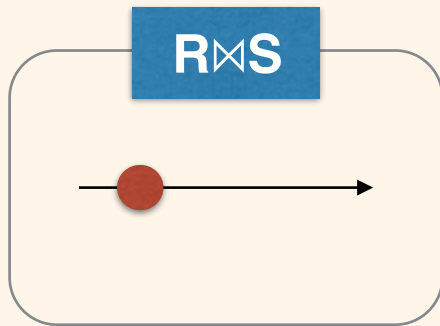
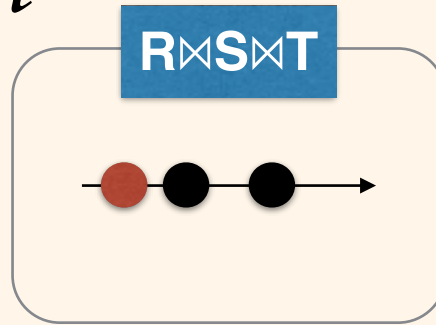
Query:  $R \bowtie S \bowtie T$





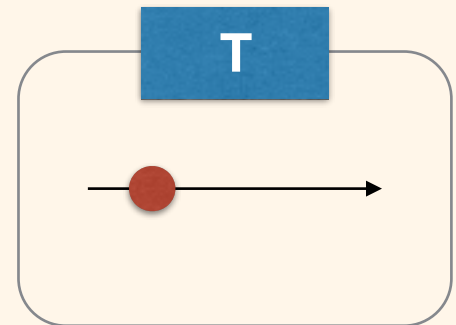
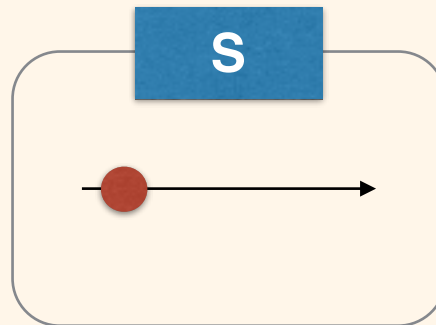
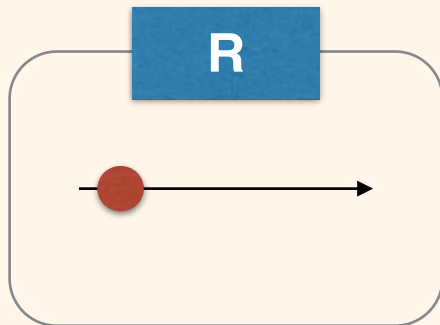
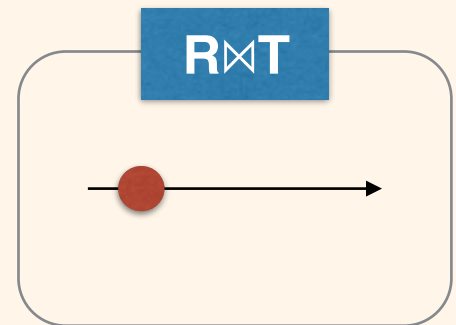
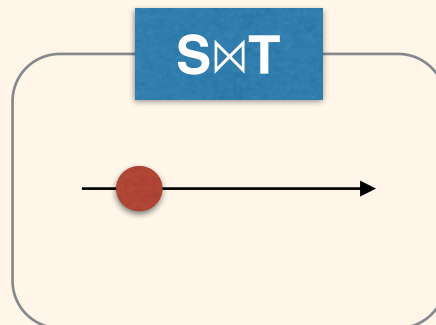
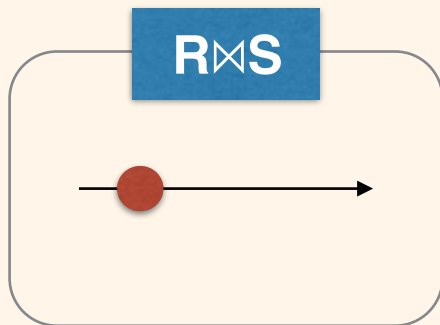
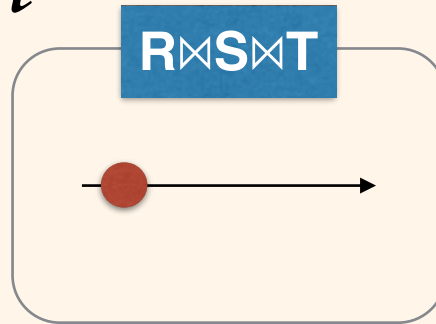
# Optimization

Query:  $R \bowtie S \bowtie T$



# Optimization

Query:  $R \bowtie S \bowtie T$

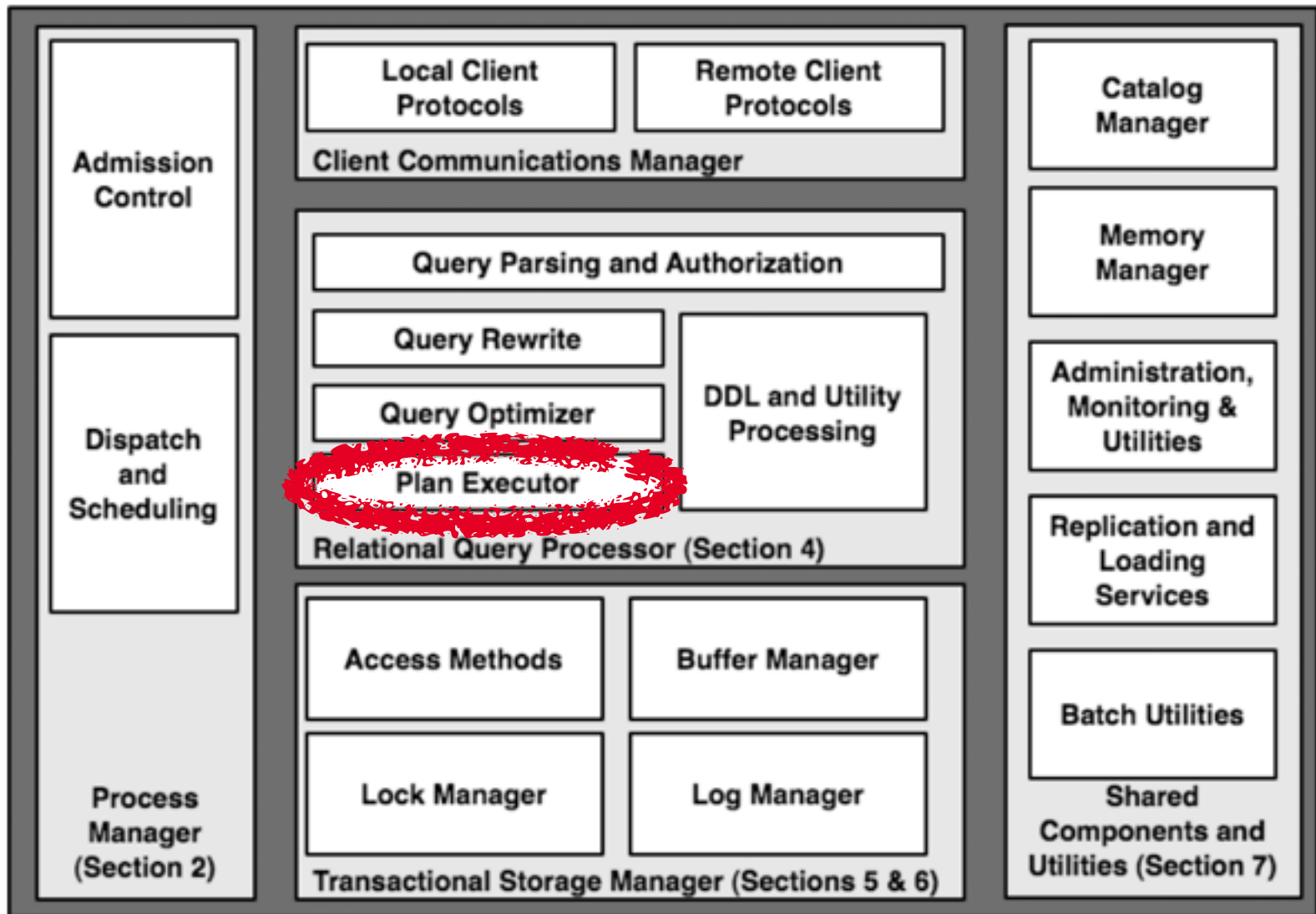




# Enumeration of Plans

- Pass 1: Find best 1-relation plan for each relation
  - ☐ includes any selects/projects just on this relation.
- Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. (*All 2-relation plans.*)
- Pass k: Find best way to join result of a (k-1)-relation plan (as outer) to the kth relation. (*All k-relation plans.*)

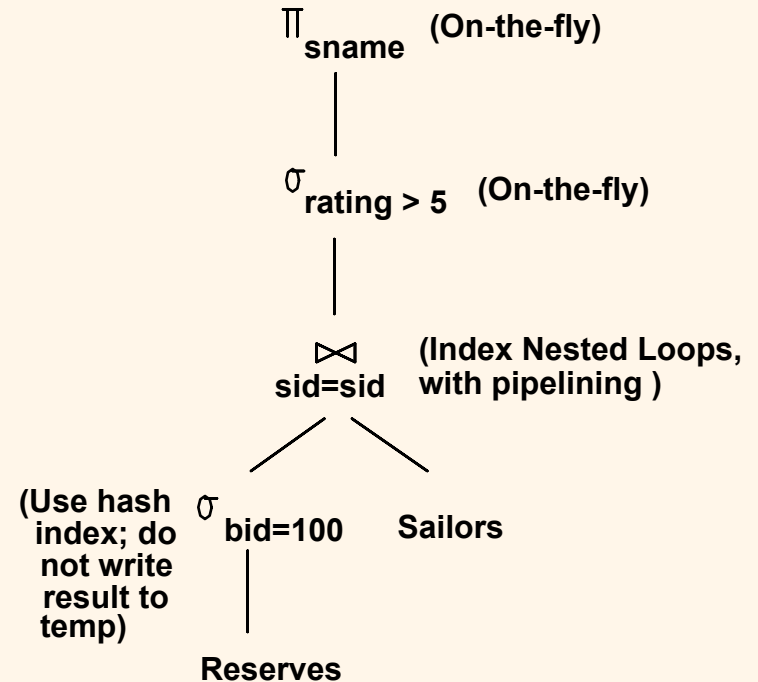
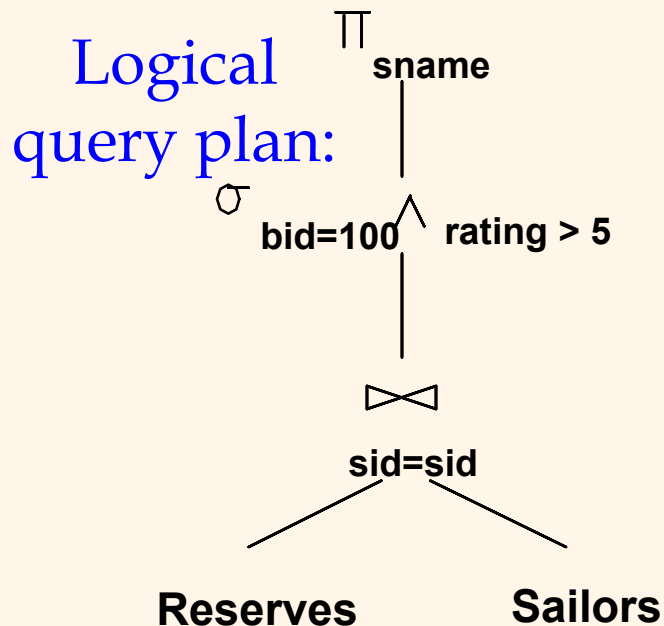
# Overview



# Query & logical and physical plans

```

SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
R.bid=100 AND S.rating>5
    
```



**[?] Physical query plan** = RA tree annotated with info on access methods and operator implementation



# *Tuple Nested Loop Join*

```
foreach tuple r in R do
  foreach tuple s in S do
    if r.sid == s.sid then add <r, s> to result
```

- ❑ R is “outer” relation
- ❑ S is “inner” relation



# *Page Nested Loop Join*

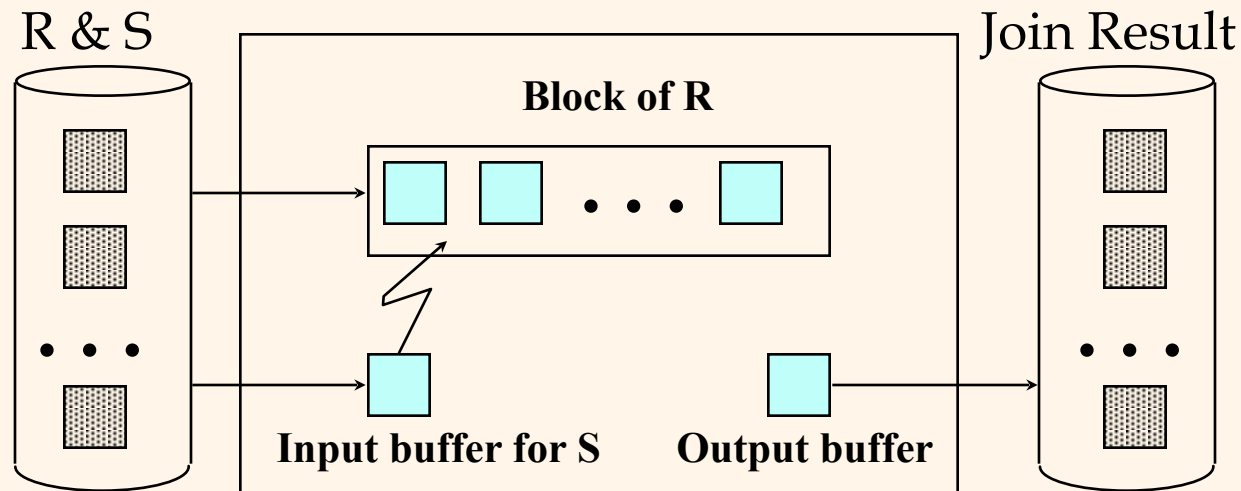
```
foreach page p1 in R do
  foreach page p2 in S do
    foreach r in p1 do
      foreach s in p2 do
        if r.sid == s.sid then add <r, s> to result
```

- ❑ R is “outer” relation
- ❑ S is “inner” relation

# Block Nested Loops Join

☐ Use one page as input buffer for scanning S, one page as output buffer, and all remaining pages to hold “block” of R.

- For each matching tuple  $r$  in R-block,  $s$  in S-page, add  $\langle r, s \rangle$  to result. Then read next R-block, scan S, etc.



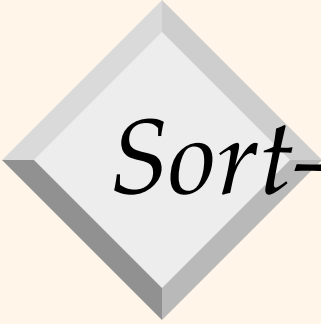




# *Index Nested Loops Join*

```
foreach tuple r in R do
    foreach tuple s in S where  $r_i == s_j$  do
        add  $\langle r, s \rangle$  to result
```

- ❑ Suppose we have an index on S, on the join attribute
- ❑ No need to scan all of S – just use index to retrieve tuples that match this r
- ❑ This will probably be faster, especially if there are few matching tuples and the index is clustered

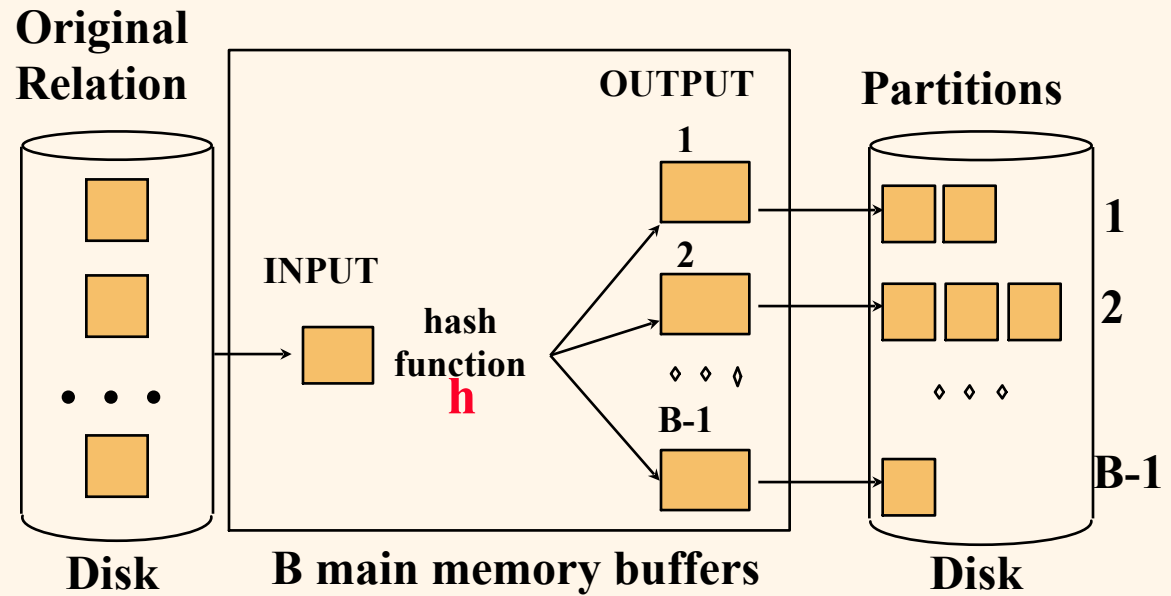


# *Sort-Merge Join*

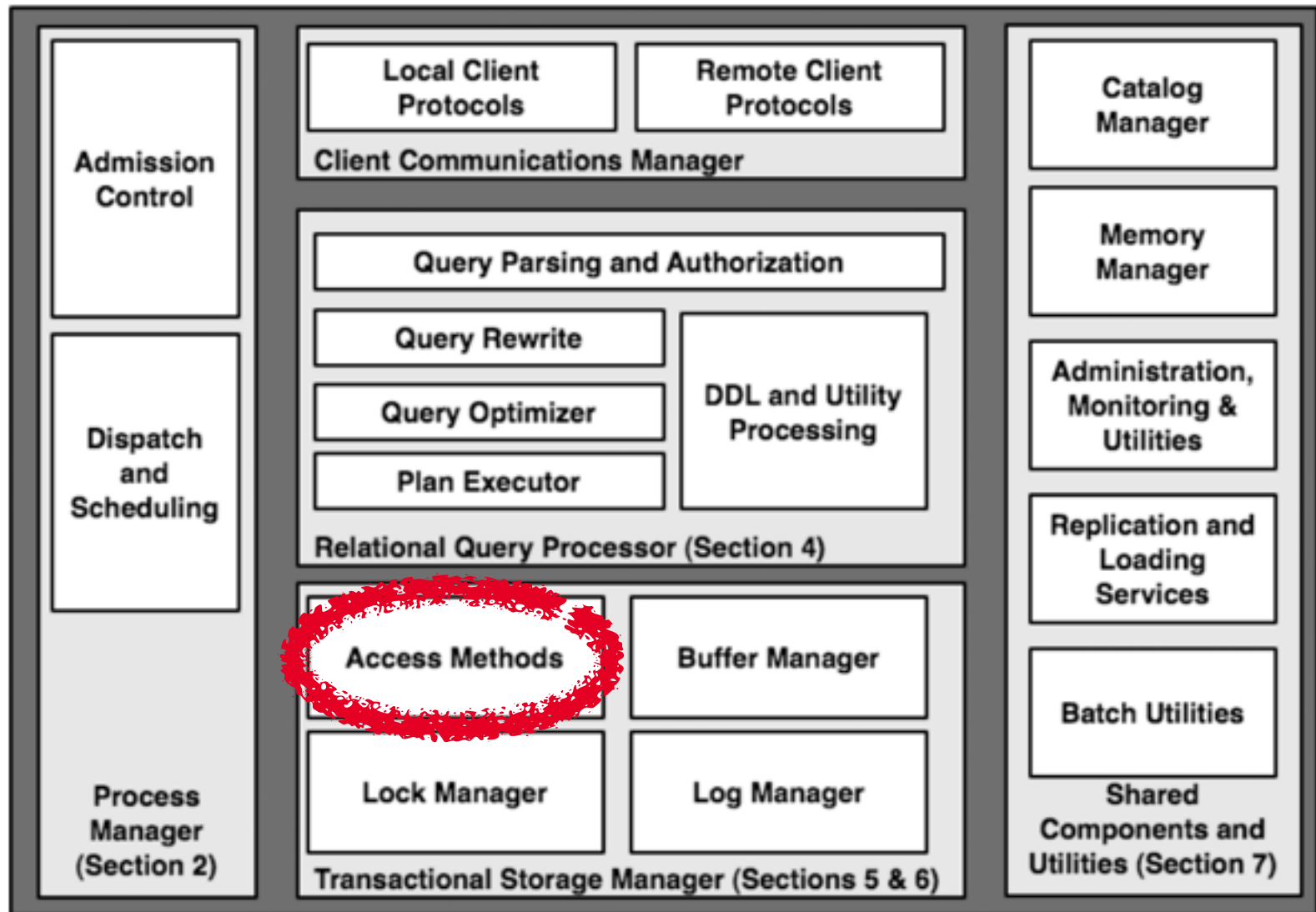
- ☐ Sort R and S on the join column, then scan them to do a “merge” (on join col.), and output result tuples.


# Hash Join

Partition both relations using hash fn  $h$ :  $R$  tuples in partition  $i$  will only match  $S$  tuples in partition  $i$ .



# Overview

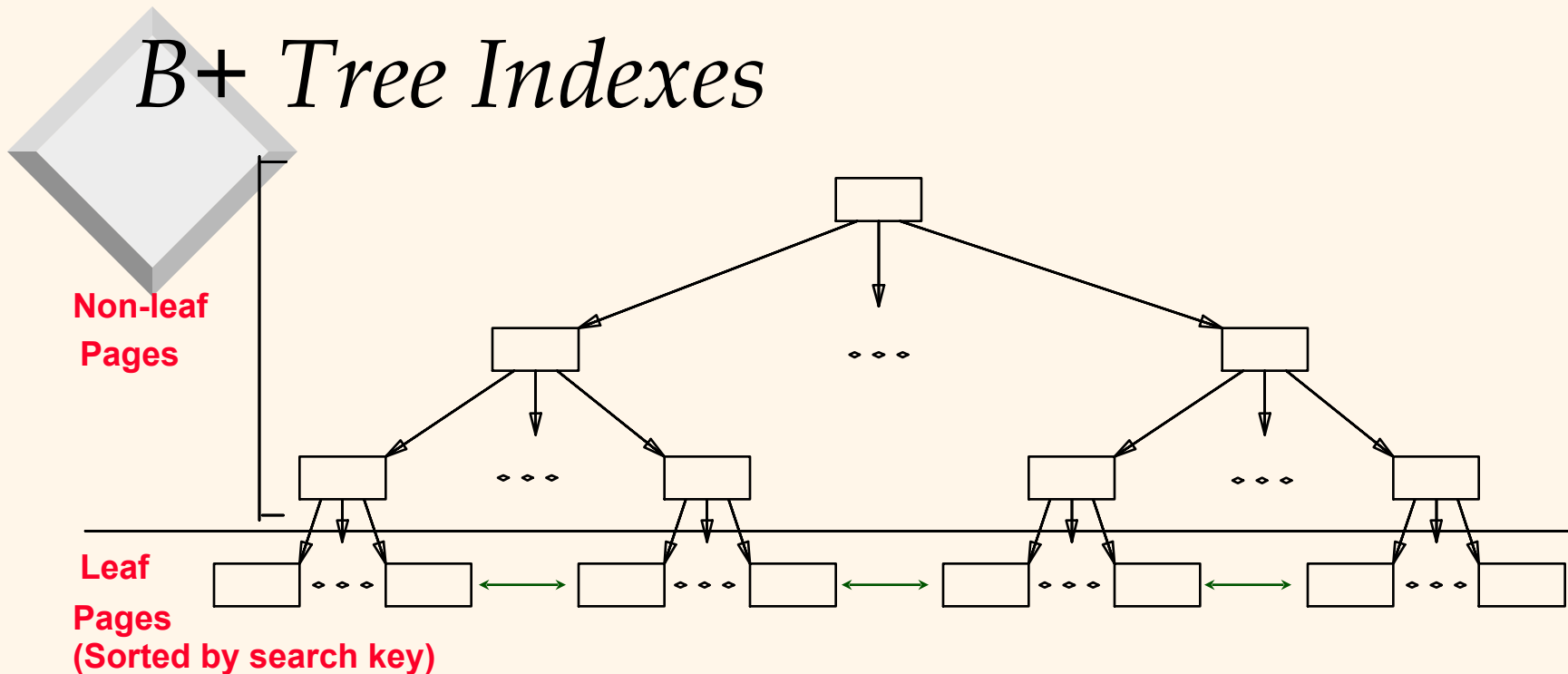




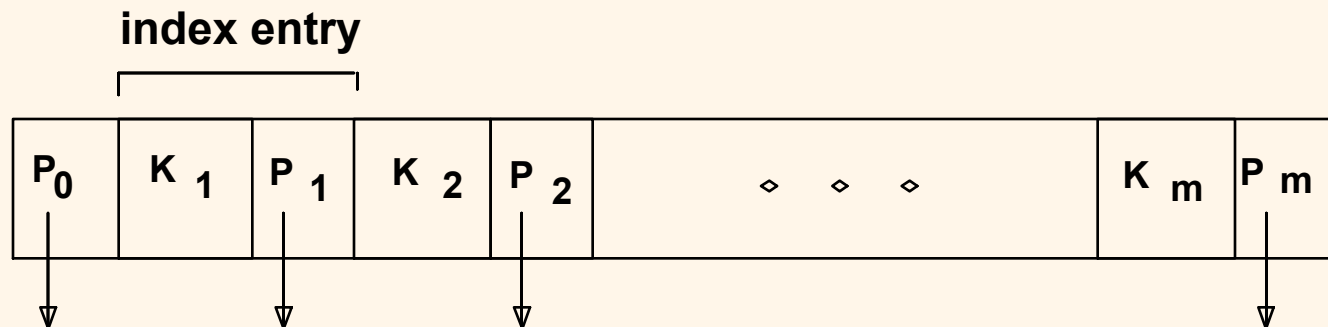
# *Tree-structured indexing*

- ❑ Tree-structured indexing techniques support both *range searches* and *equality searches*.
- ❑ ISAM: static structure; B+ tree: dynamic, adjusts gracefully under inserts and deletes.
  
- ❑ Simple cost metric for discussion of search costs: number of disk I/Os (i.e. how many pages need to be brought in from disk)
  - Ignore benefits of sequential access etc to simplify

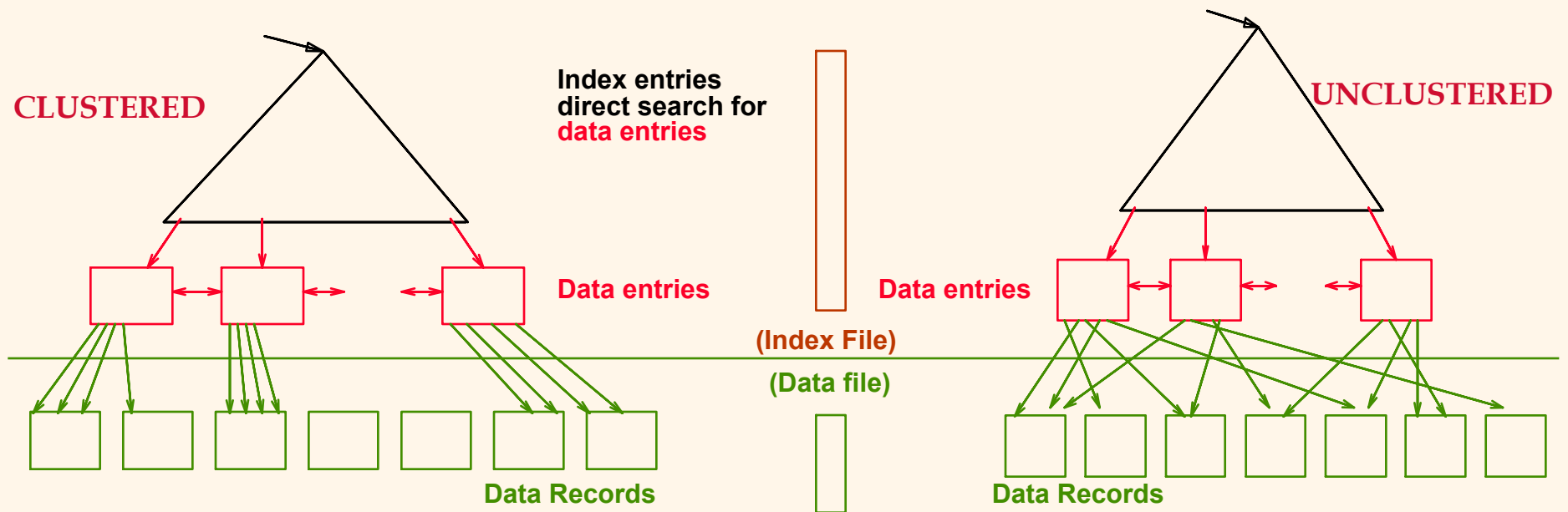
# B+ Tree Indexes




- ❖ Leaf pages contain *data entries*
- ❖ Non-leaf pages have *index entries*; only used to direct searches:



# Clustered vs. Unclustered Index



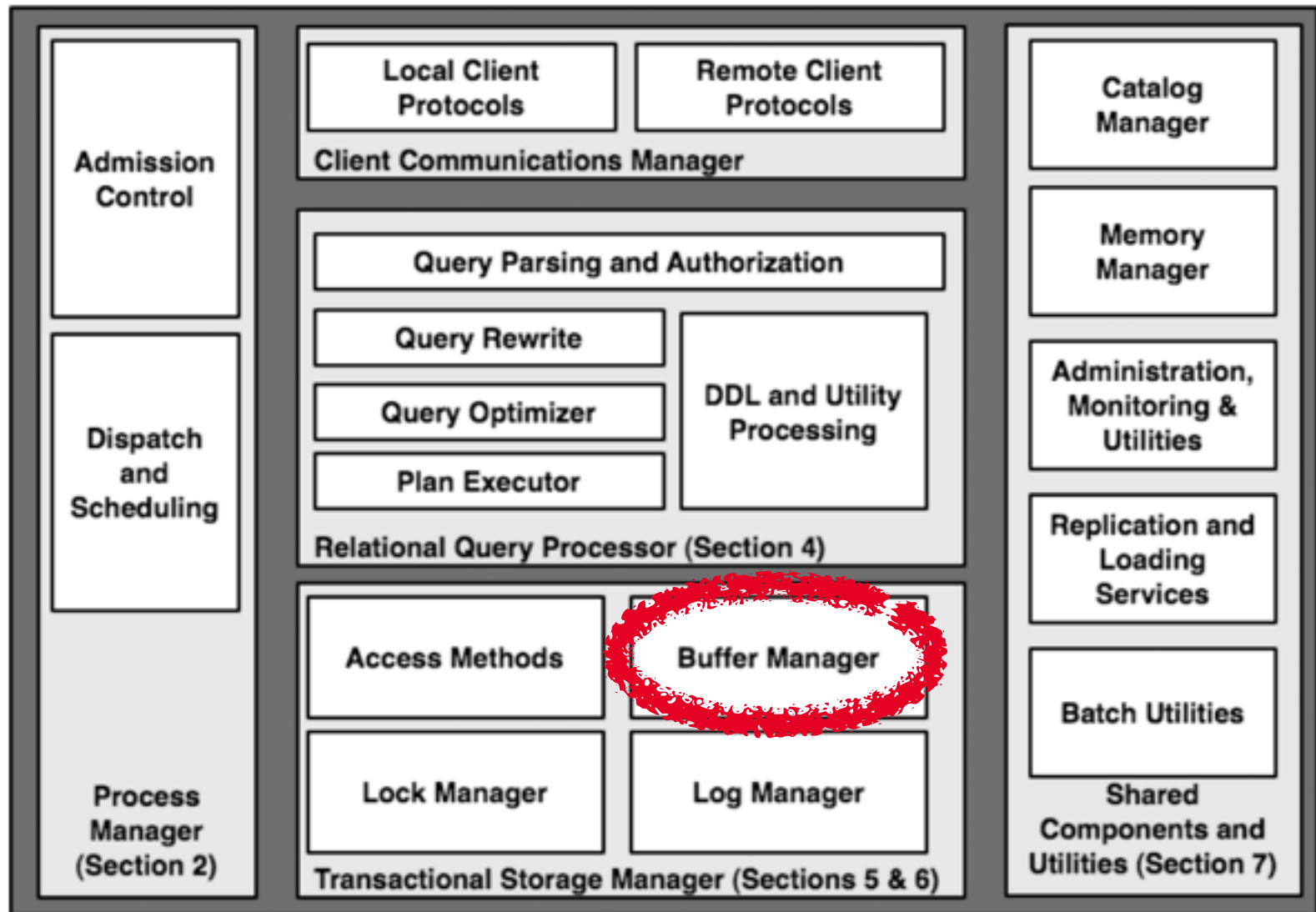


# *Indexing using Hashing*

- ❑ Hash-based indexes are for *equality selections*. **Cannot** support range searches.
- ❑ Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.

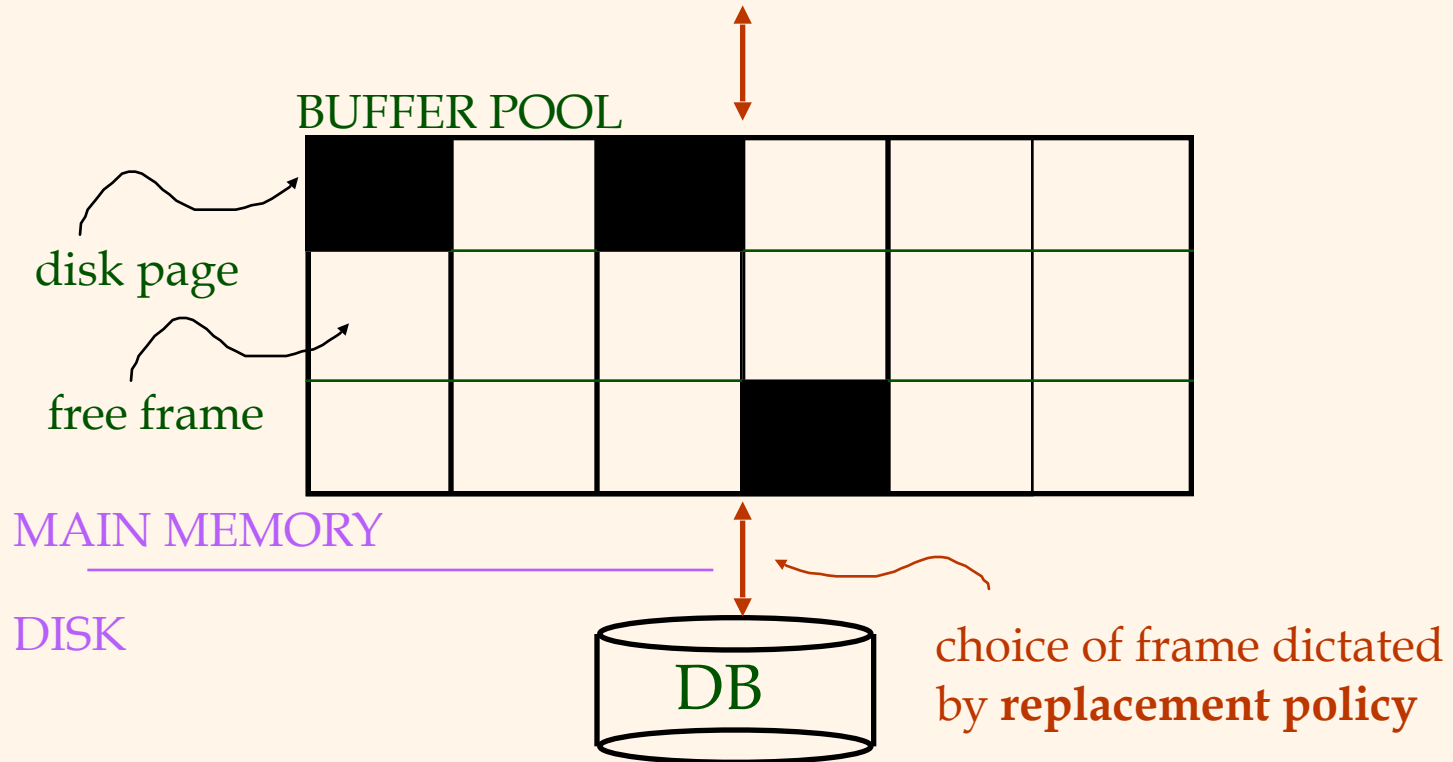


# Overview



# Buffer Management in a DBMS

Page Requests from Higher Levels



❓ Data must be in RAM for DBMS to operate on it!

❓ Table of <frame#, pageid> pairs is maintained.

# When a Page is Requested ...

☐ If page is not in pool (cache miss):

- Choose a frame for replacement
- If frame contains a page with changes, write it to disk
- Read requested page into chosen frame
- *Pin* the page and return its address.

☐ If requested page is in pool (cache hit):

- Increment its pin count and return its address.

☐ *If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time*



# *Buffer Replacement Policies*

❑ Lots of other replacement policies:

❑ MRU

❑ LFU (Least Frequently Used)

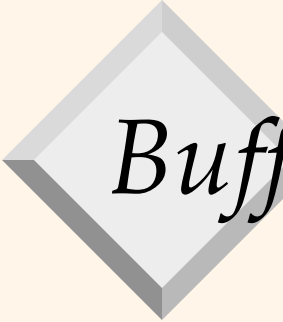
❑ Random

❑ FIFO (First In First Out)

❑ Clock (Round Robin)

❑ Different benefits for different workloads

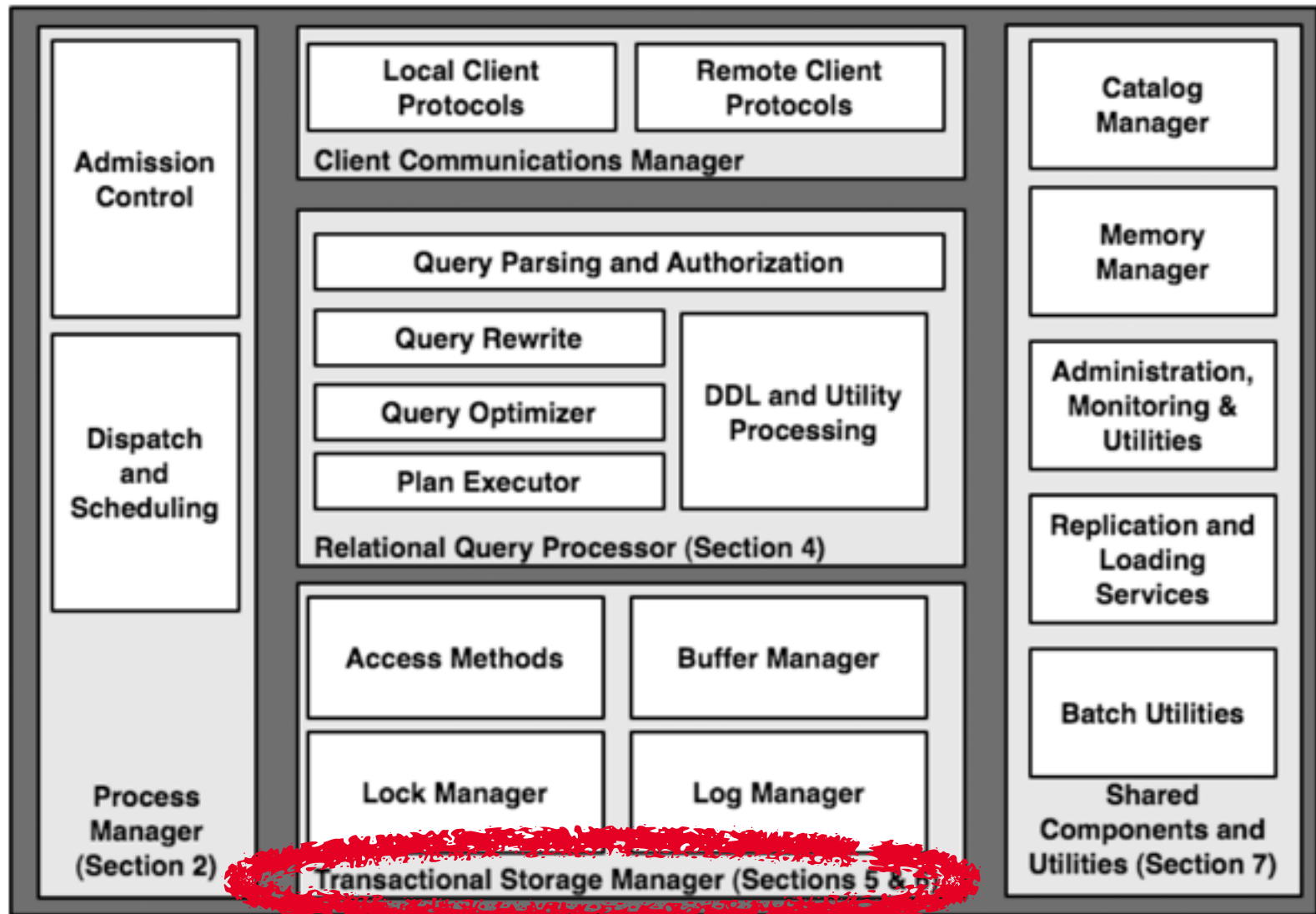
❑ Also, some require keeping less state than others



# Buffer Replacement Policy (Contd.)

- ❑ Policy can have big impact on # of I/O's; depends on the *access pattern*.
- ❑ Sequential flooding: Nasty situation caused by LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O.
  - Example scenario: join implementation with nested loops

# Overview



# Transactions

- ☐ Are a fundamental database abstraction
- ☐ ACID properties
  - Atomicity
  - Durability
  - Consistency
  - Isolation
- ☐ Broadly supported in relational DBMSs
- ☐ NoSQL support is a moving target

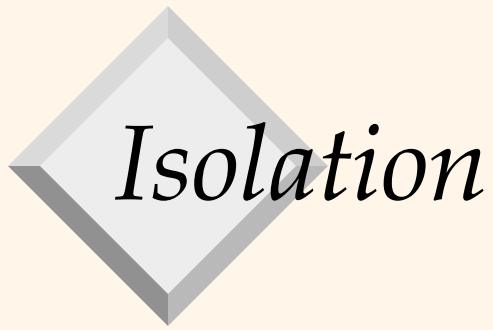
# Atomicity

- ❑ A transaction should execute completely or not at all
- ❑ If the first few statements succeed, but the next one fails, the entire transaction must be **rolled back**
  - This failure could be due to an error/exception or to a system crash
- ❑ It ain't over till it's over – nothing is guaranteed until the transaction **commits**



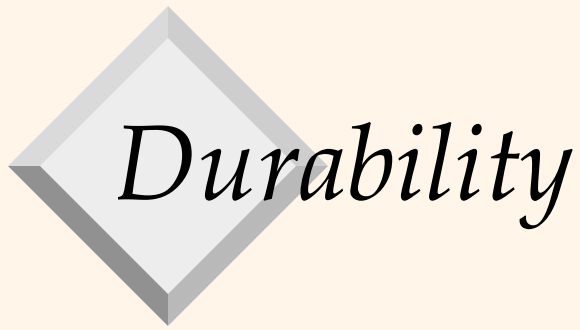
# Consistency

- ☐ Assume we have an intrinsic notion of **data consistency**
  - E.g. semantic constraints are satisfied by DB
    - ☐ E.g. every order has associated billing info
- ☐ The "C" in ACID: A transaction, if executed by itself on a consistent DB, will produce another consistent DB
  - An assumption that a transaction is a self-contained unit of work (no loose ends)



# *Isolation*

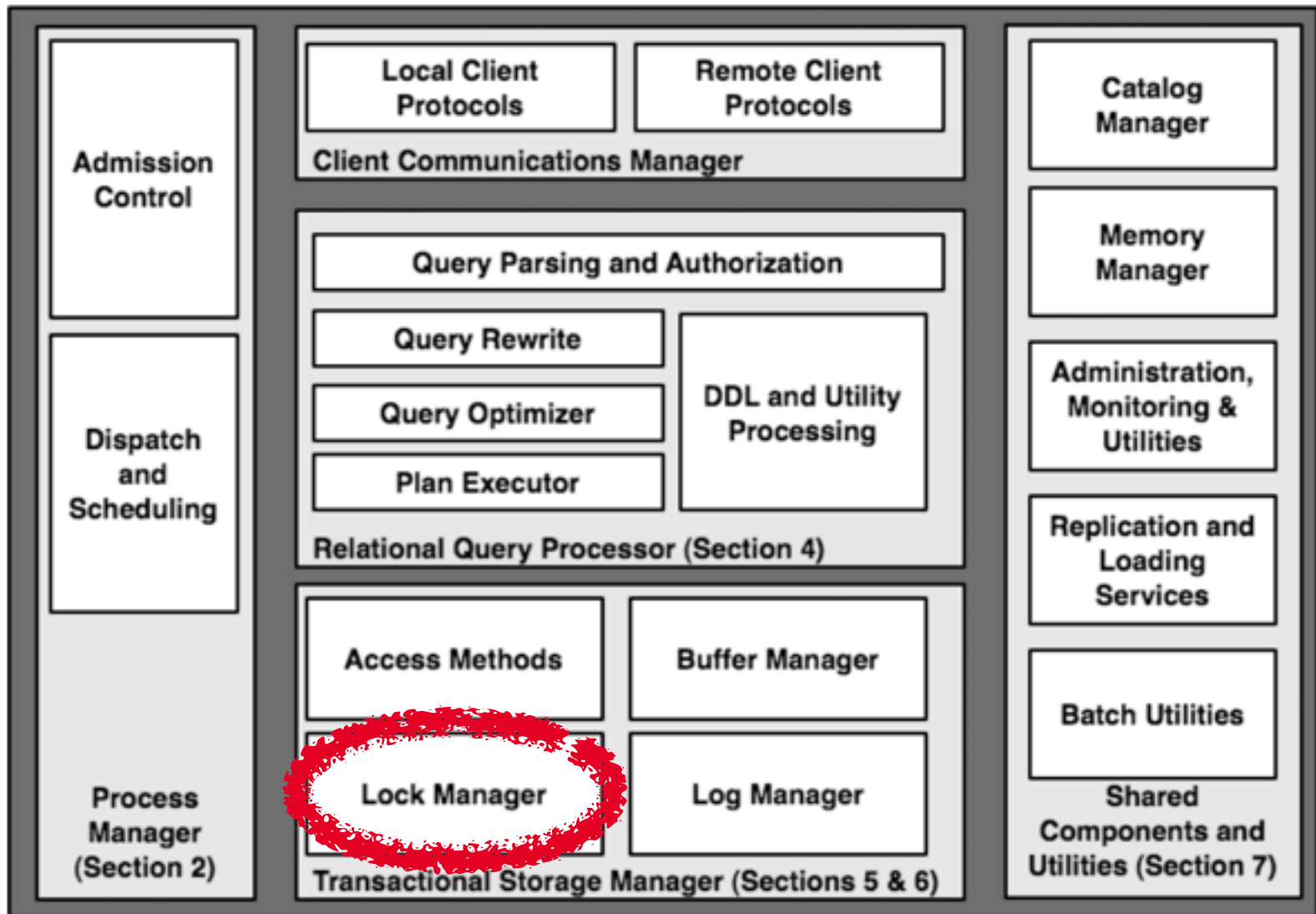
- ❑ No harmful interference between transactions is permitted as they run
- ❑ Every transaction should have the illusion of having the DB to itself



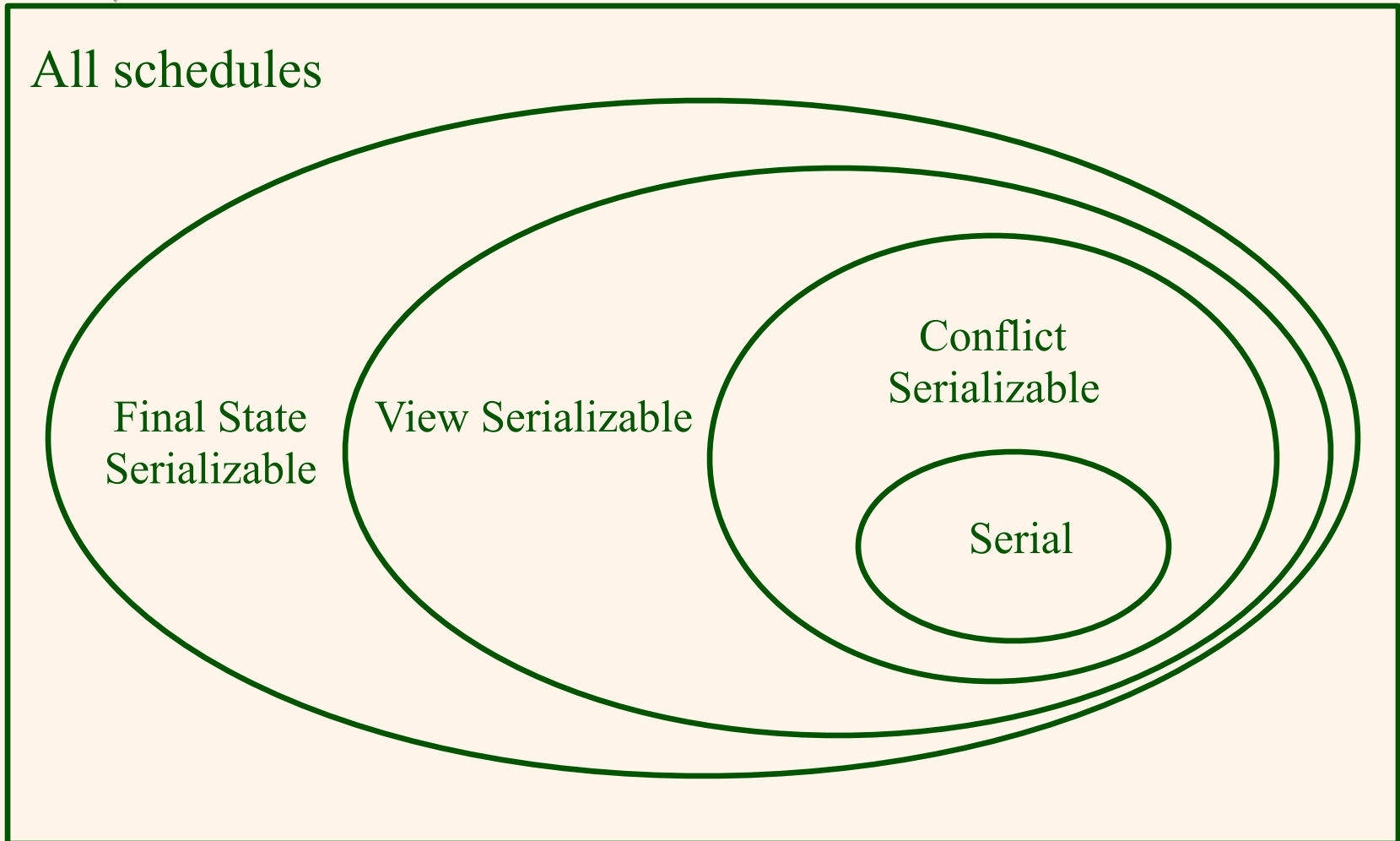
# *Durability*

- ❑ Once a transaction does **commit**, the changes should be **persistent**
- ❑ If system crashes before changes make it to disk, this could be a problem!
- ❑ Does not preclude the ability to "undo" a real world action, e.g. cancel an order
  - But this must be done using a second transaction.

# Overview



# *Big Picture (all inclusions are proper)*



# Conflict Graphs

- ❑ Given a schedule, can identify all conflicting pairs of operations and represent them as a graph
- ❑ Nodes are transactions
- ❑ Edge from  $i$  to  $j$  if transaction  $i$  contains an operation that **conflicts with and precedes** (in the schedule) an operation by transaction  $j$
- ❑ Example:  $R1(A)$   **$W2(A)$**   $R1(A)$



# *Conflict Serializability*

- ❑ A schedule is conflict serializable if its conflict graph contains no cycle
- ❑ Alternative (equivalent) statement: it is conflict serializable if it has the same conflict graph as some serial schedule
  - Why are these equivalent?
- ❑ Topological sort on the conflict graph gives us equivalent serial execution

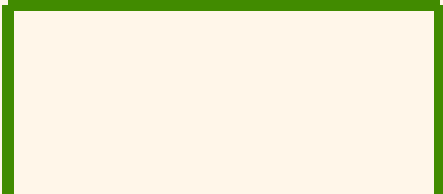
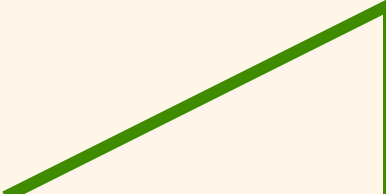
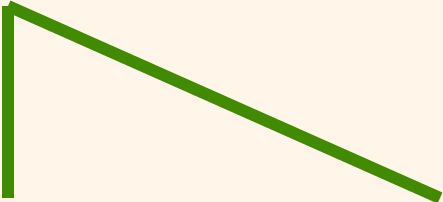
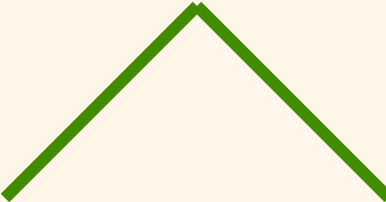


# *Locking-Based Protocols*

- ☐ First family of protocols – based on idea of locks
- ☐ Before any read or write, a transaction must request a lock on an object
  - A "permission to operate" on this object
- ☐ Locks are managed centrally by the DBMS lock manager



# 2PL variants

<i>Conservative</i> <i>Strict</i>	Yes	No
Yes		
No		

# *Optimistic CC*

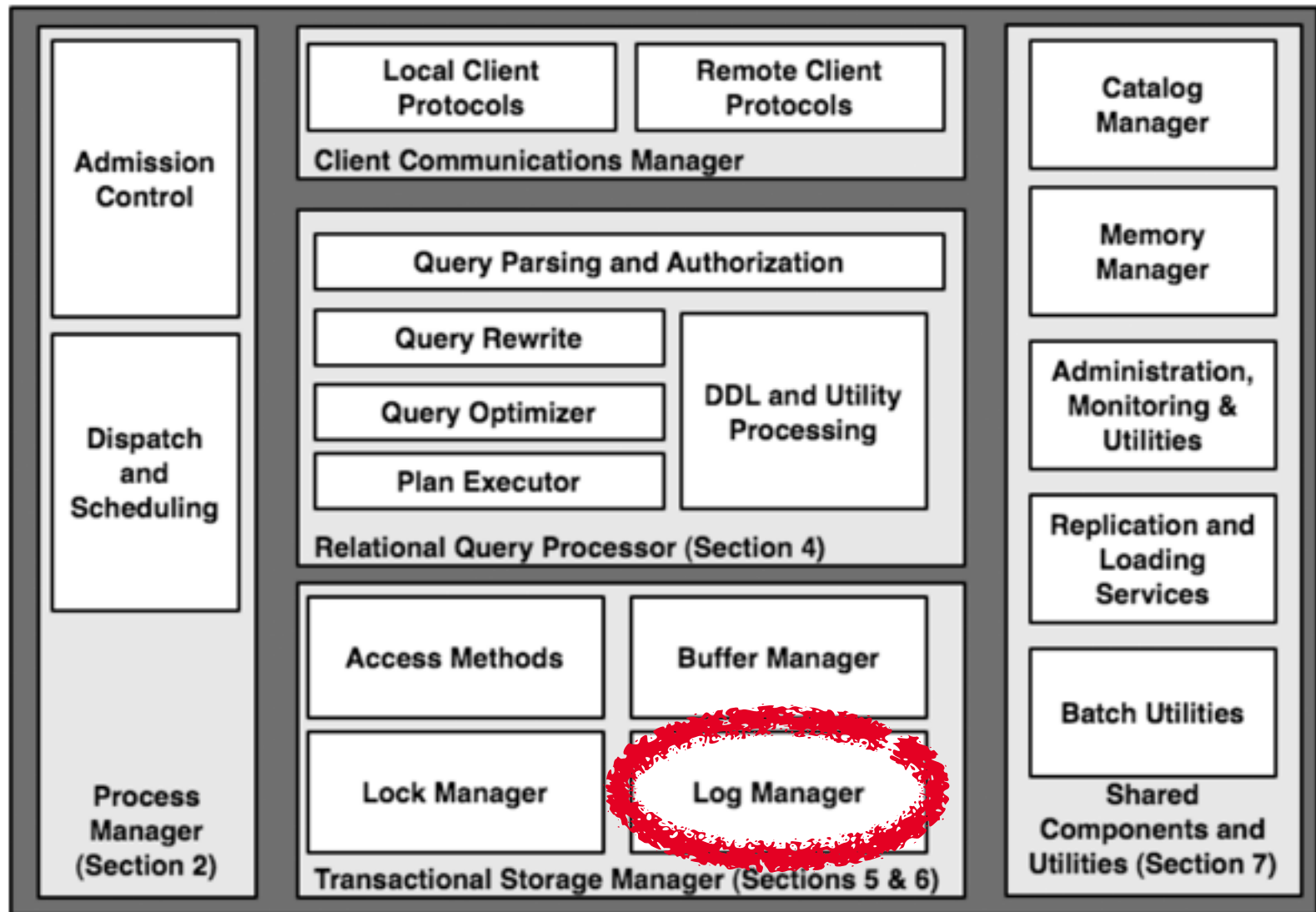
- ❑ Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - These overheads occur even if conflicts are rare
- ❑ If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before commit.



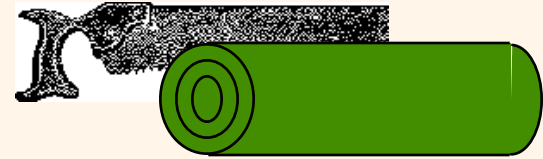
# MVCC

- ❑ System keeps several versions of each data item
- ❑ When a transaction writes a data item, it creates a new version rather than overwriting
- ❑ When a transaction reads a data item, the version visible to the read is determined by the protocol used (several options)
- ❑ Maintaining versions can be nontrivial and comes with its own extra cost, of course

# Overview



# Basic Idea: Logging



- ❑ Record REDO and UNDO information, for every update, in a *log* that will survive crashes.
  - Log is written sequentially.
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.
- ❑ **Log:** An ordered list of REDO/UNDO actions
  - Log record contains:
    - <transID, pageID, offset, length, old data, new data>
  - and additional control info (which we'll see soon).



# *Write-Ahead Logging (WAL)*

## ☐ The Write-Ahead Logging Protocol:

- Must force the log record for an update before the corresponding data page gets to disk.
- Must write all log records for a transaction before commit.

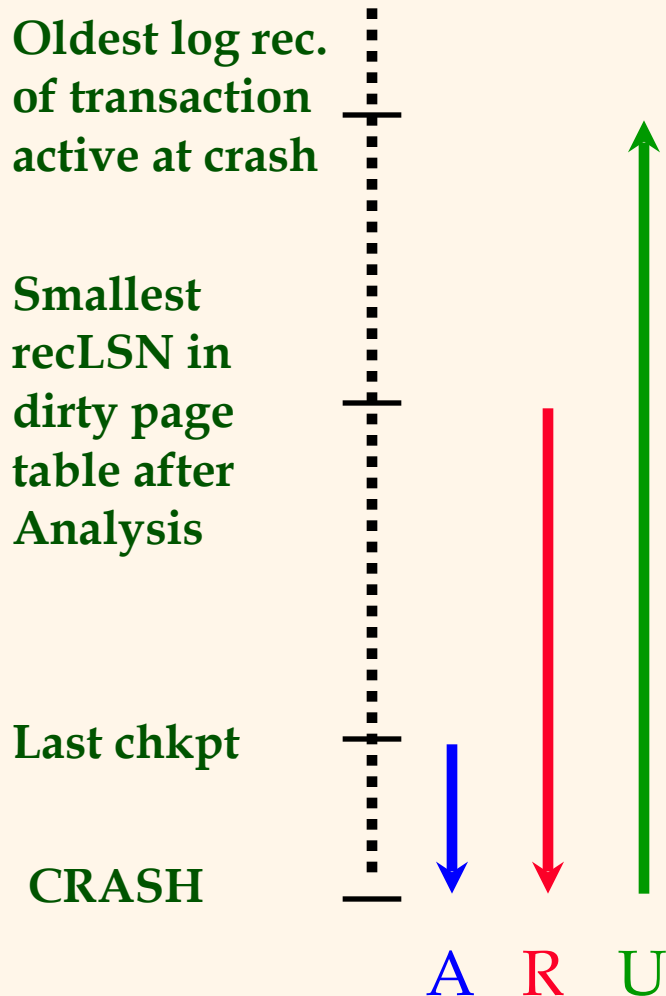
☐ #1 guarantees Atomicity (why?)

☐ #2 guarantees Durability (why?)

☐ Exactly how is logging (and recovery!) done?

- We'll study the **ARIES** algorithm.

# Crash Recovery: Big Picture



☐ Start from a checkpoint (found via master record).

☐ Three phases. Need to:

- Figure out which transactions committed since checkpoint, which failed (*Analysis*).
- **REDO** *all* actions.
  - ☐ (repeat history)
- **UNDO** effects of failed transactions.

# Overview

