

# Database Management Systems 2

## A Systems View

Final Version WS 2013/2014

Prof. Dr. Alejandro Buchmann  
DVS - Datenbanken und Verteilte Systeme  
Fachbereich Informatik, TUD  
Hochschulstr. 10  
[buchmann@informatik.tu-darmstadt.de](mailto:buchmann@informatik.tu-darmstadt.de)

# Administrivia

-  Overview
-  DBMS architecture revisited
-  RAID

- Office hours: Monday 16:30-18:00
- Please make an appointment or ask after class if office hours will be held
  - Maria Tiedemann 166230
- Problem session (Übung) to be held by Robert Gottstein
  - office hours: W 16:00-17:30 or by appointment
  - First week of class we will have a lecture during problem session hours
- Information available on our Web-page
  - Username: db2                      Password: JMS@db2

# Course Objectives

-  Overview
-  DBMS architecture revisited
-  RAID

- Using a relational DBMS in a *naive* manner is easy
- Using a DBMS *effectively* requires understanding of the DBMS's internals
- Understanding how a DBMS functions is not trivial
- This course is about the *fun* (and *hard work*) of *understanding* the *principles* behind a DBMS, how it is *implemented*, and how it is *optimized*.
- The goal is to enable you to administer, tune, optimize, and intelligently exploit database systems

# Course Overview

Overview

DBMS architecture revisited

RAID

- DBMS architecture revisited
- Storage media and hierarchy
- Database engineering rules of thumb
- Buffer management
- Record, page, and file formats and organization
- Access paths and indexing
- Implementation of relational operators
- Query optimization
- (Extended) transaction models
- Transaction Processing
- Concurrency control
- Recovery
- Cluster architectures (e.g. RAC, Services and Clustering, Oracle Streams)
- Effects of new HW and trends
- Other topics that I will add as time permits (e.g. NoSQL, main memory DBs, multitenant DBMSs, etc.)



# Main References

- Ramakrishnan, R., Gehrke, J.  
*Database Management Systems*  
2<sup>nd</sup> Ed., McGraw Hill, 2000  
3<sup>rd</sup> Ed., McGraw Hill, 2003 rev. 2007
- Bernstein, P., Hadzilacos, V., Goodman, N.  
*Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987 (out of print, available on-line courtesy of Phil Bernstein)
- Weikum, G., Vossen, G.  
*Transactional Information Systems - Theory, Algorithms, and the Practice of Concurrency Control and Recovery*  
Morgan Kaufmann, 2002
- Elmasri, R., Navathe, S.  
*Fundamentals of Database Systems*, Addison Wesley, 6<sup>th</sup> Ed., 2010

# Main References

- Tom White  
*Hadoop: The Definitive Guide, O'Reilly, 3<sup>rd</sup> Ed., 2012*
- Pramod J. Sadalage, Martin Fowler  
*NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, Pearson Education, Aug 2012*
- Gray, J., Prashant, S.  
*Rules of Thumb in Data Engineering, Proc. 16th Intl. Conf. on Data Engineering, San Diego, Feb. 2000*
- Martin L. Abbott, Michael T. Fisher  
*Scalability Rules: 50 Principles for Scaling Web Sites, 2011*

# Main References (cont.)

- Gray, J., Reuter, A.  
*Transaction Processing: Concepts and Techniques*  
Morgan Kaufmann, 1993
- Härdler, T., Rahm, E.  
*Datenbanksysteme - Konzepte und Techniken der Implementierung*  
Springer, 1999
- Graefe, G.  
*Query evaluation techniques for large databases*  
ACM Comp. Surv., 25:2, 1993, pp. 73-170
- Tumma, M.  
*Oracle Streams: High Speed Replication and Data Sharing*  
Rampant Techpress, 2005

# Main References (cont.)

- Vallath, M.; *Oracle 10g RAC: Grid, Services & Clustering*, Digital Press, 2006.
- Niemiec, Richard; *Oracle Database 11g Release 2 Performance Tuning*, Oracle Press
- Bryla, Bob, Loney, Kevin; *Oracle Database 12c, The Complete Reference*, Oracle Press, Sept. 2013
- Gray, J., Prashant, S.; *Rules of Thumb in Data Engineering*  
Proc. 16th Intl. Conf. on Data Engineering, San Diego, Feb. 2000
- Petrov, I., Gottstein, R., Ivanov, T., Bausch, D., Buchmann, A.;  
*Page Size Selection for OLTP Databases on SSD RAID Storage*  
Journal of Information and Data Management, Vol. 1, No. 2, 2011

# Main References (cont.)

- Bausch, D., Petrov, I., Buchmann, A.;  
*On The Performance Of Database Query Processing Algorithms On Flash Solid State*, Workshop on Flexible Database and Information System Technology, 2011
- Oracle Real Application Clusters (RAC)  
[http://en.wikipedia.org/wiki/Oracle\\_RAC](http://en.wikipedia.org/wiki/Oracle_RAC)

# Main References (cont.)

- RAID [http://www.usbyte.com/common/raid\\_systems.htm](http://www.usbyte.com/common/raid_systems.htm)
- RAID Levels 0+1 (01) and 1+0 (10)  
<http://www.pcguide.com/ref/hdd/perf/raid/levels/multLevel01-c.html>
- RAID Levels 0+3 (03 or 53) and 3+0 (30)  
<http://www.pcguide.com/ref/hdd/perf/raid/levels/multLevel03-c.html>
- Basic functioning of disk  
<http://www.usbyte.com/common/HDD3.htm#Basic>
- P.M.Chen, E.K.Lee, G.A.Gibson, R.H.Katz, D.A.Patterson  
*RAID: High Performance*  
Reliable Secondary Storage, ACM Computing Surveys, Vol.26(2),  
pp.145-185, June 1994. (classic paper)

# NOTE / WARNING / ALERT

-  Overview
-  DBMS architecture revisited
-  RAID

The database area, although mature, is dynamic. I will update the course as we proceed during the semester

New transparencies and references will be added, topics may change!

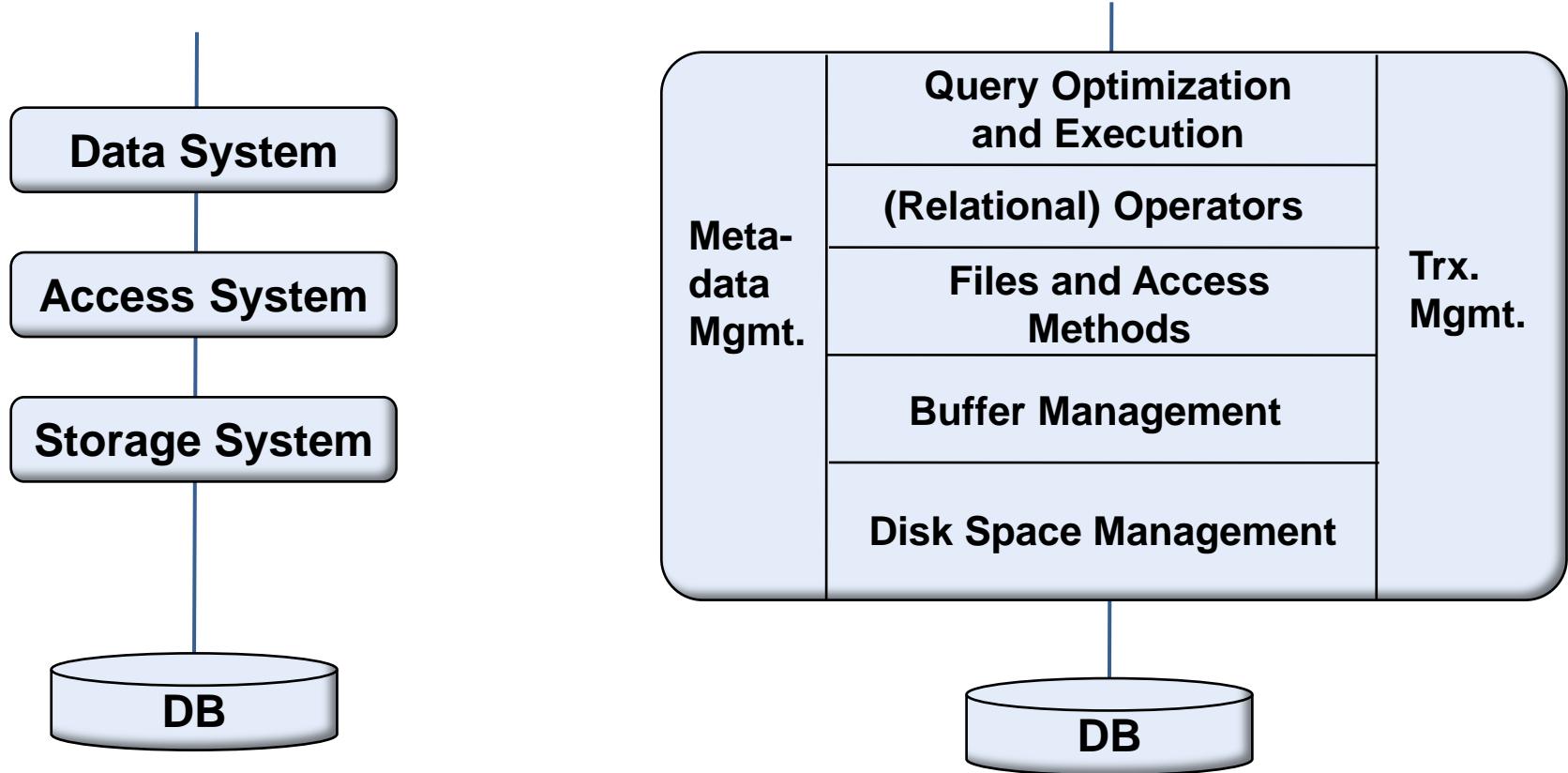
Some changes occurred as we abandoned Informix as primary industrial reference system and started to look closer at Oracle. PostgreSQL is a great open source system to play with

New chapters on concurrency control, databases on flash

Use last year's slides as reference and to make annotations but make sure to download the final version of the slides at the end of the semester. You will be held responsible at exam time for the material that is current at time of taking an exam.



# Simplified Architectures



Overview

DBMS architecture revisited

RAID

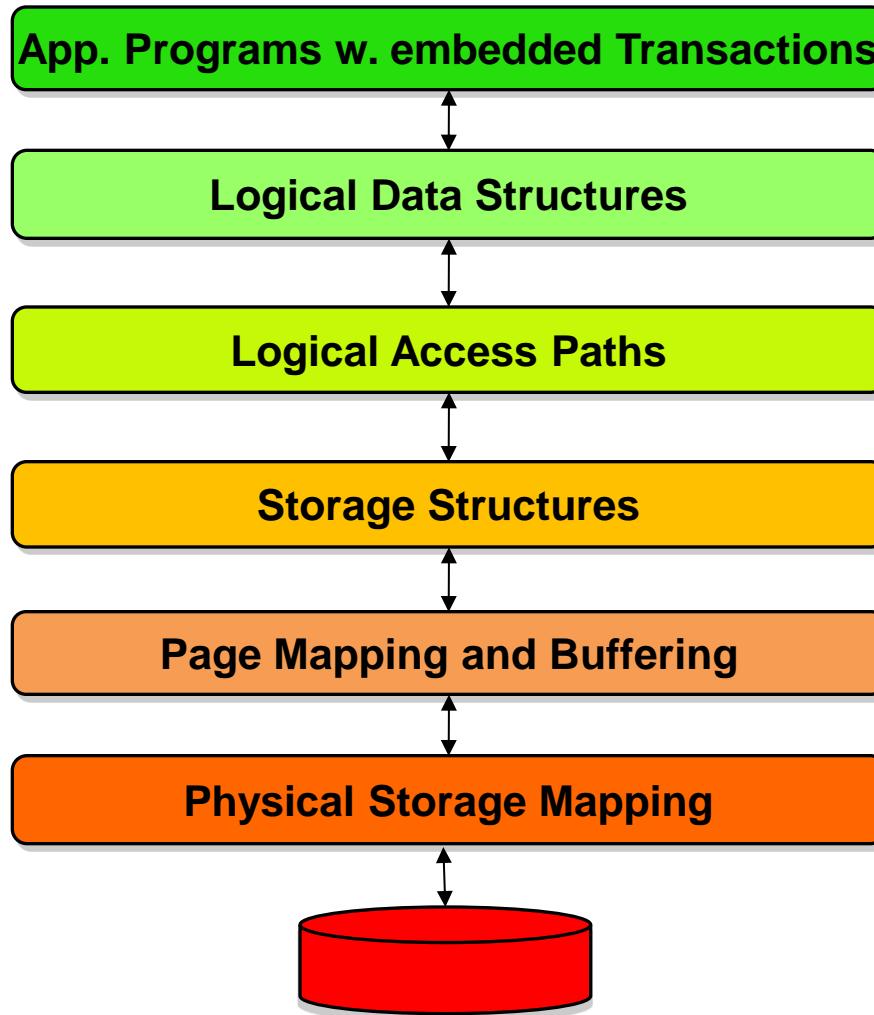


# Layering Principle

- Represent system as a hierarchy of layers  $I_0, \dots, I_n$  [Senk75, Parn75]
- System-layer  $i$  is described in terms of the elements of layer  $i-1$
- Each virtual machine is implemented with the abstractions provided by the VM at the lower level
  - limited number of layers (5-7) - layer simplicity vs. efficiency
  - generic description of functionality
  - implementation independence
- Härdér's layer architecture [Härd83, HR99] - good for teaching purposes, too rigid for implementation

# Layer Architecture

-  Overview
-  DBMS architecture revisited
-  RAID



**Set-oriented interface, declarative QL**

**Record-oriented interface**

**Internal interface**

**DB-buffer interface**

**File interface**

**Device interface**

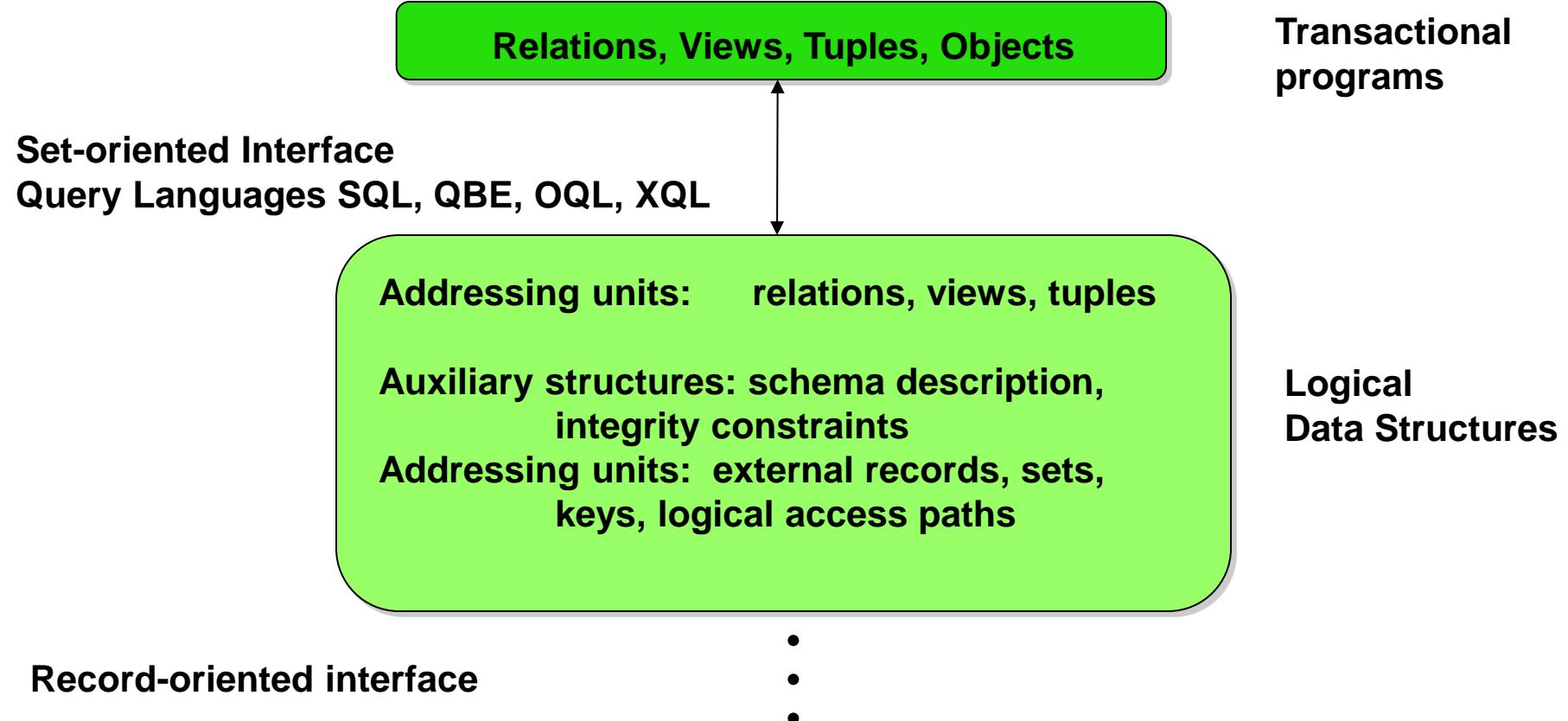


# Transaction Programs and Logical Datastructures

Overview

DBMS architecture revisited

RAID



# Logical Data Structures and Logical Access Paths

Overview

DBMS architecture revisited

RAID

Logical  
Data Structures

Addressing units: relations, views, tuples

Auxiliary structures: external schema descr.,  
integrity constraints

Addressing units: external records, sets,  
keys, logical access paths

Record-oriented interface

FIND <record name>

STORE <record name>

Logical  
Access Paths

Addressing units: external records, sets,  
keys, logical access paths

Auxiliary structures: access paths, internal  
schema description

Addressing units: internal records, B-trees,  
hash-tables, etc.

Internal interface



# Logical Access Paths and Storage Structures

Overview

DBMS architecture revisited

RAID

## Logical Access Paths

- Addressing units:** external records, sets, keys, logical access paths
- Auxiliary structures:** access paths, internal schema description
- Addressing units:** internal records, B-trees, hash-tables, etc.

Internal interface  
store record, update  
index, etc.

## Storage Structures

- Addressing units:** internal records, B-trees, hash-tables, etc.
- Auxiliary structures:** address tables, page tables, free page tables
- Addressing units:** pages, segments

DB-buffer interface

⋮



# Storage Structures and Page Mapping/Buffering

Overview

DBMS architecture revisited

RAID

Storage  
Structures

DB buffer interface  
get page i  
free page j

**Addressing units:** internal records, B-trees, hash-tables, etc.

**Auxiliary structures:** address tables, page tables, free page tables

**Addressing units:** pages, segments

Page Mapping  
Structures

**Addressing units:** pages, segments

**Auxiliary structures:** page tables, block tables, segment definitions, etc.

**Addressing units:** blocks, files

File interface



# Page Mapping/Buffering - Physical Storage Mapping

Overview

DBMS architecture revisited

RAID

File interface  
read block i  
write block k

Device interface  
channel programs

**Addressing units:** pages, segments

**Auxiliary structures:** page tables, block tables  
segment definitions, etc.

**Addressing units:** blocks, files

**Addressing units:** blocks, files

**Auxiliary structures:** free-space info., extent  
tables, file directories

**Addressing units:** tracks, cylinders,  
channels

**Page Mapping  
Structures**

**Physical  
Storage  
Mapping**

**Devices**



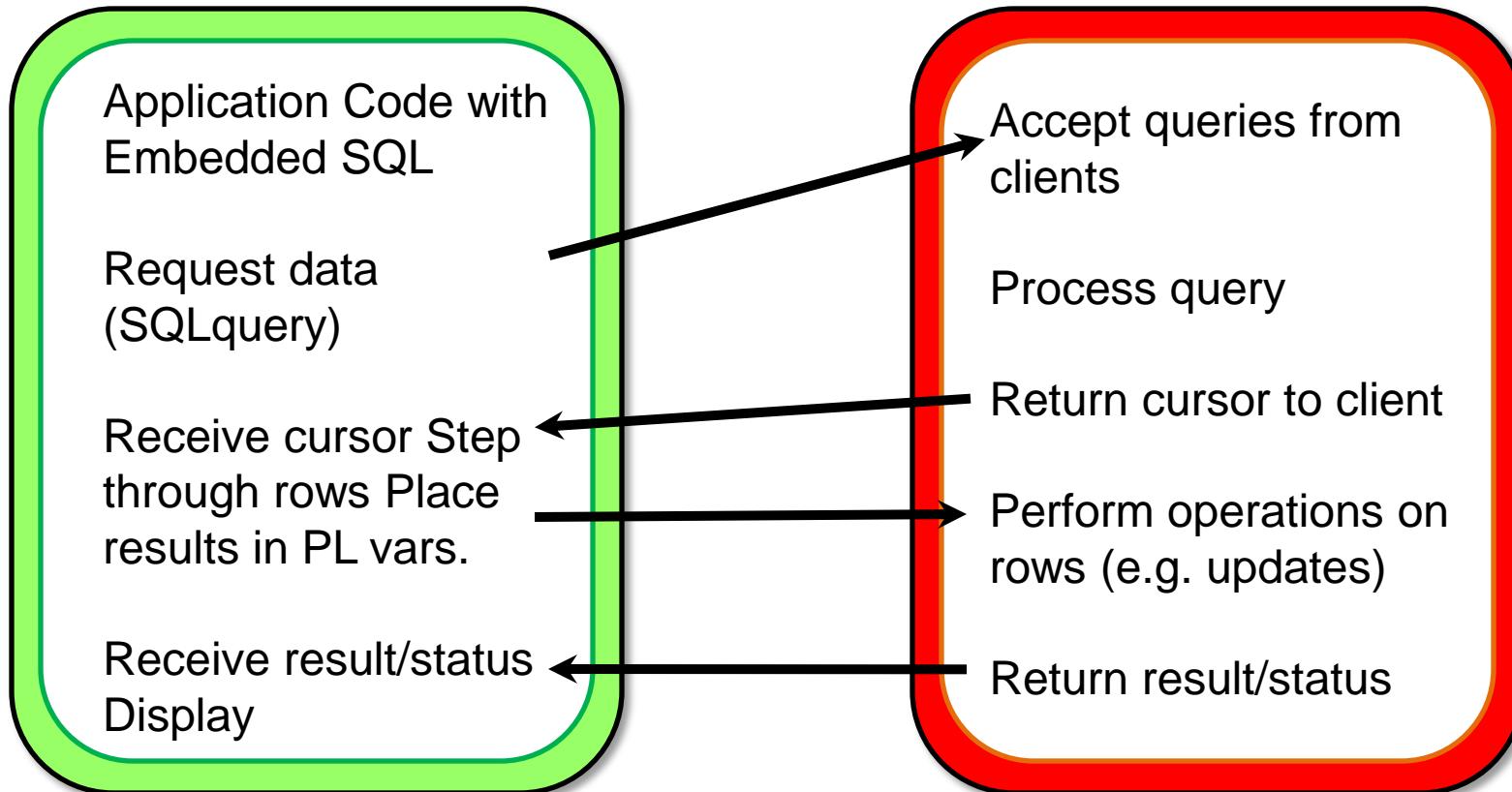
# Architectures in Client/Server Environments

Overview

DBMS architecture revisited

RAID

## Relational C/S DBMS based on query shipping



# Object Oriented (or XML-based\*) C/S Architectures

Overview

DBMS architecture revisited

RAID

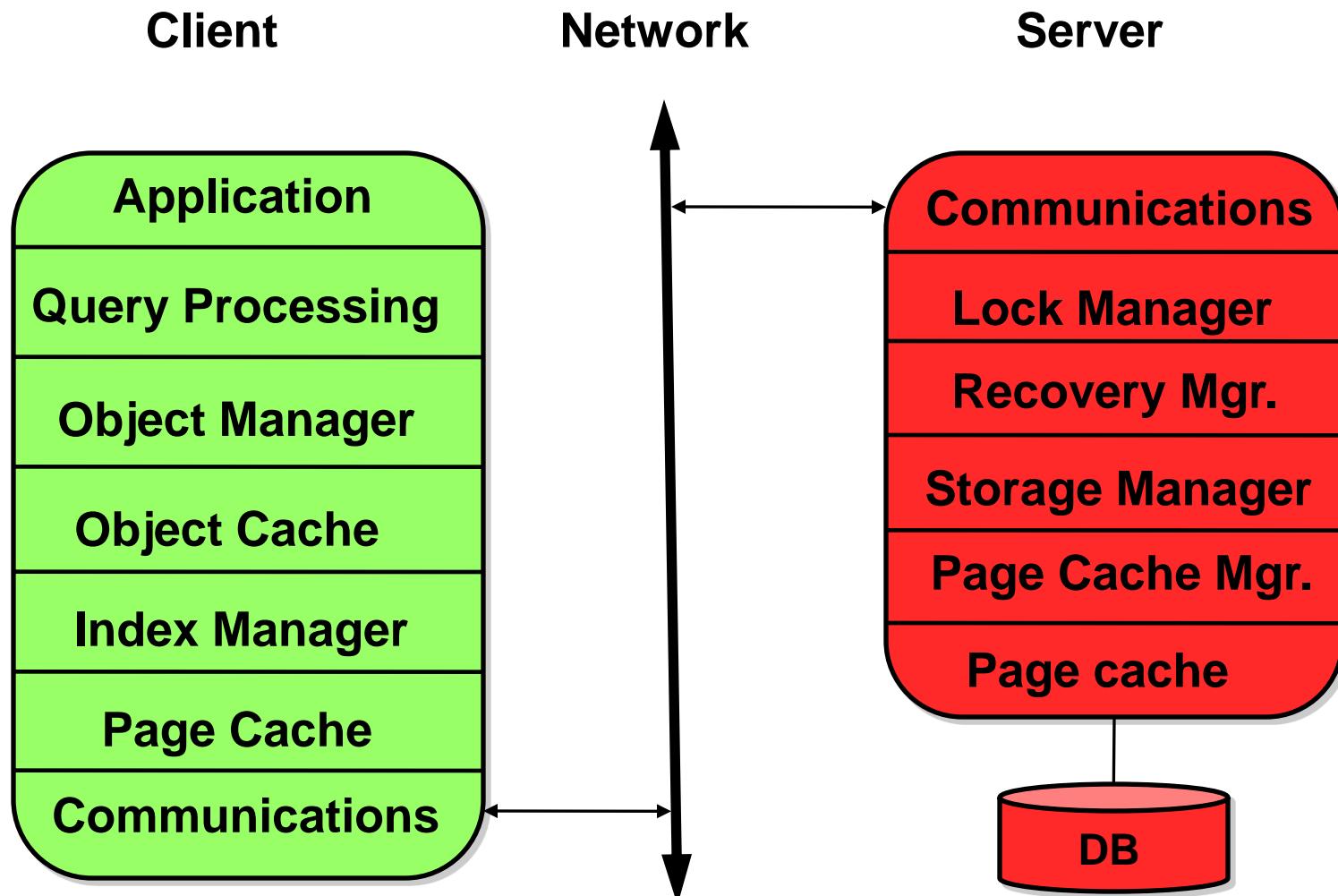
- Separation of functionality
- Motivation: high compute power on desktop (typically underused), impedance mismatch
- Fuzzy separation of DB-query and application program
- Associative and navigational access
- Different distribution of functionality and transfer unit results in different base architectures:
  - page server
  - object server

\* Note: Many XML-DBMSs are based on OODBMS engines or proprietary mechanisms for document storage based on OODBMS principles

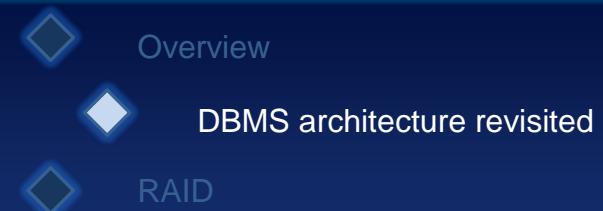


# Page Server Architecture

-  Overview
-  DBMS architecture revisited
-  RAID



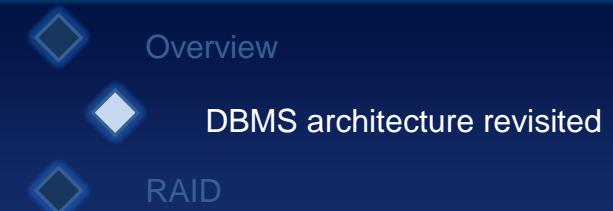
# Page Server Architecture



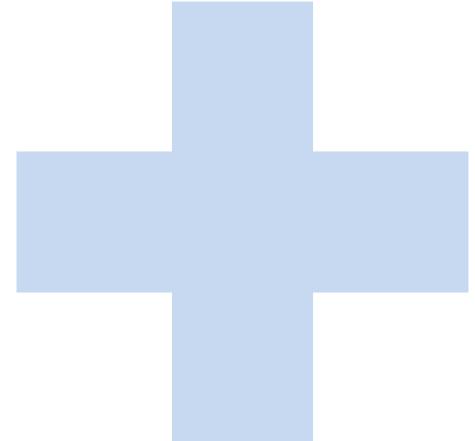
- Transfer unit is a page
- Server concentrates on disk I/O, buffering pages, locking and logging
- Methods can only be executed on client
- Objects on client-side can be kept packed in page cache or unpacked in object cache



# Page Server Architecture: Advantages



- Better load distribution between clients and server
- Server can be optimized for efficient page access, recovery and concurrency control
- Transferring a 4KB page is only marginally more expensive than sending individual objects
- Good clustering minimizes transfer cost
- Single server can serve more clients



# Page Server: Disadvantages

- Methods can only be executed on client
- Integrity constraints only executable on clients may result in large additional page transfers
- Object locks are hard to implement
- Performance critically depends on clustering
- Clustering for fast access may conflict with concurrency requirements resulting in blocking

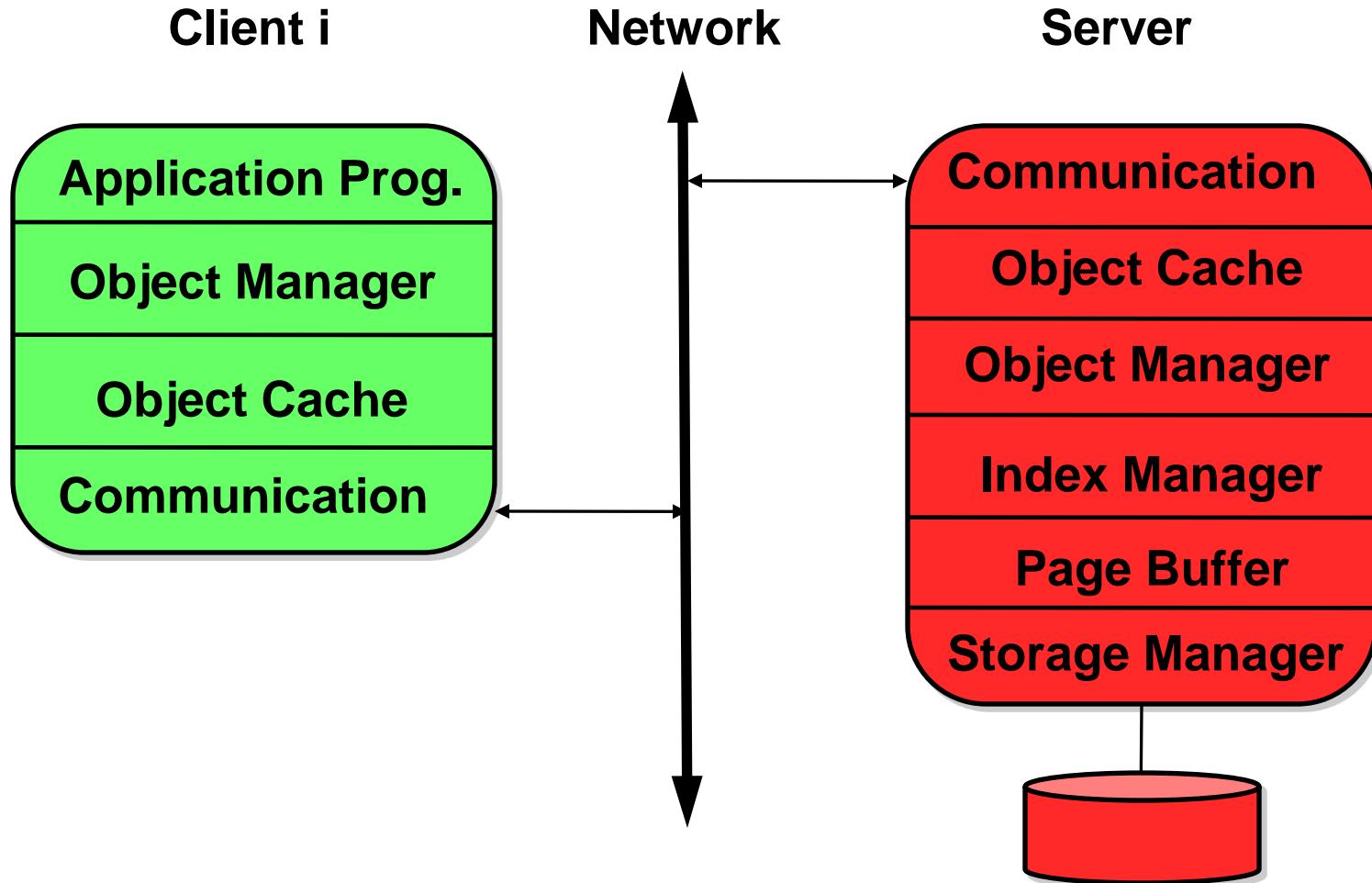


# Object Server Architecture

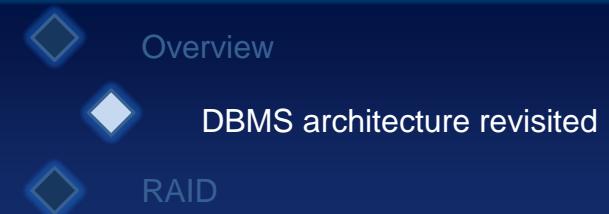
Overview

DBMS architecture revisited

RAID



# Object Server Architecture



- Transfer unit is object
  - DBMS functionality replicated on server and client, i.e. server unpacks objects
  - Both client and server keep object cache
- Search sequence
  - local object cache
  - RPC to server
  - server object cache
  - server page buffer
  - disk

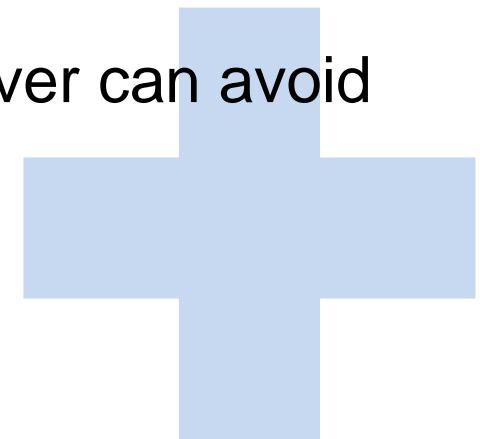
# Object Server Architecture: Advantages

Overview

DBMS architecture revisited

RAID

- Both server and client can execute methods (in principle, in case of compiled languages server only accesses state in today's implementations)
- Easy implementation of multi-object integrity constraints (language dependent)
- Easy implementation of object locks
- In case of small intermediate results server can avoid unnecessary transfers



# Object Server Architecture: Disadvantages

Overview

DBMS architecture revisited

RAID

- 1 RPC/object (worst case) if object not in client object cache
- Objects may be scattered in multiple places: client object cache(s), server cache(s), disk
- Methods executing on server must collect objects
- Page locks harder to implement



# Food for thought

- OODBMSs disappeared for all practical means and purposes
  - How is the object-relational mismatch handled today?
  - How and where do ORMs enter the picture?
  - What is Hibernate and how does it work?
- What is the difference between an XML-enabled DBMS and a native XML-DBMS?

<http://www.rpbourret.com/xmlXMLDatabaseProds.htm#native>



# Multiprocessor Architectures

- Three base architectures
  - *shared nothing* - good for distribution, partitioning/replication
  - *shared disk* - all disks reachable from each processor
  - *shared everything* - OS offers single address space abstraction, commonly accessible main memory and disks
- Oracle RAC is realized on multiprocessor systems according to the shared everything architecture
  - Multiple copies of the DBMS but cache consistency guaranteed through cache fusion (high speed interconnect process-to-process communication)
- Other systems, such as DB2 and MS-SQL Server are implemented in shared nothing mode

# Cloud Databases

- A cloud database typically runs on a cloud computing platform (Amazon EC2, Salesforce, Microsoft Azure,...)
- Two common deployment models:
  - Users deploy their own VM image
  - Users purchase access to a database service
- Two basic models
  - Relational (SQL-based, e.g. Oracle, DB2, PostgresDB...)
  - NoSQL (e.g. MongoDB, Casandra, ...)

# Deployments

- Virtual machine image
  - Cloud platforms offer virtual machine images for limited time purchase
  - Users can either upload their own DBMS image or can purchase an optimized DBMS image (e.g. Oracle 11g Enterprise Edition on Amazon EC2)
- Database as a Service (DBaaS)
  - Cloud provider installs and maintains DBMS, users can rent it as a service (e.g. Amazon's SimpleDB, Microsoft's Azure SQL Database)
- Managed database hosting on the cloud
  - Provider maintains and runs users' database on their behalf



# Cloud DB Architectures

- Web-based console for launching a DB app., making DB snapshots (backups), monitoring statistics
- Console connects to DB manager which allows the user to launch, install, remove, monitor and tune a DB.
- Database services make underlying software stack (OS, DBMS, 3<sup>rd</sup> party SW) transparent to the user
- Database services scale
  - Automatic scaling adds needed resources automatically
  - Manual scaling permits adding resources by request
  - High availability provided via SLAs (99.99%, 99.999%,...)

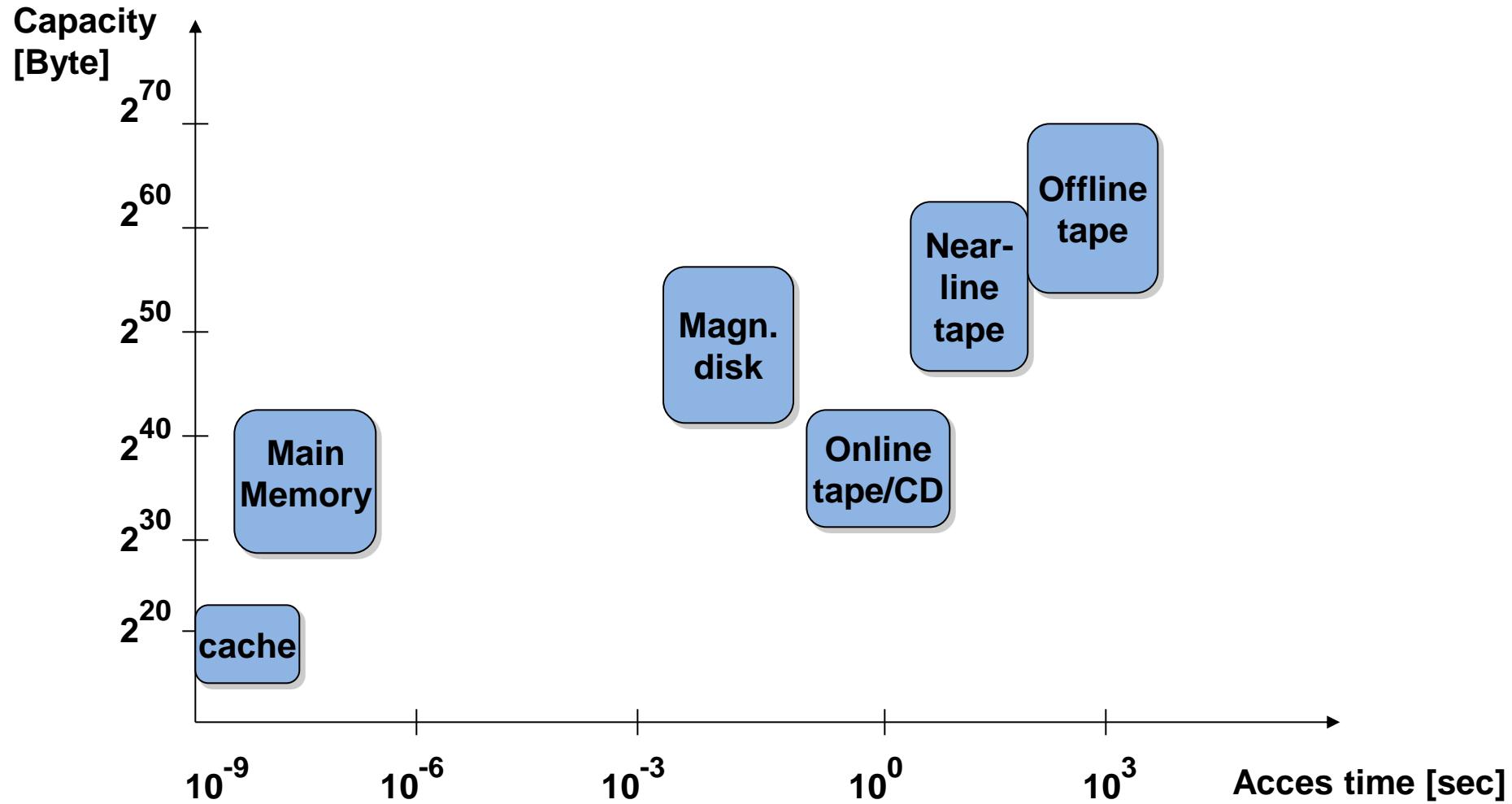
# TP light vs. TP heavy

- TP light refers to a 2 tier client/server architecture
  - Client connects directly to DBMS
  - Main bottleneck is number of DB connections available
  - For many years DBMSs could not handle more than hundreds to a few thousands of clients
- TP heavy refers to transaction processing in a 3 tier architecture
  - Client connects to a TP Monitor (nowadays application server)
  - TPM or app server maintains connections and performs load sharing
  - 3 tier architecture necessary to deal with many thousands of clients



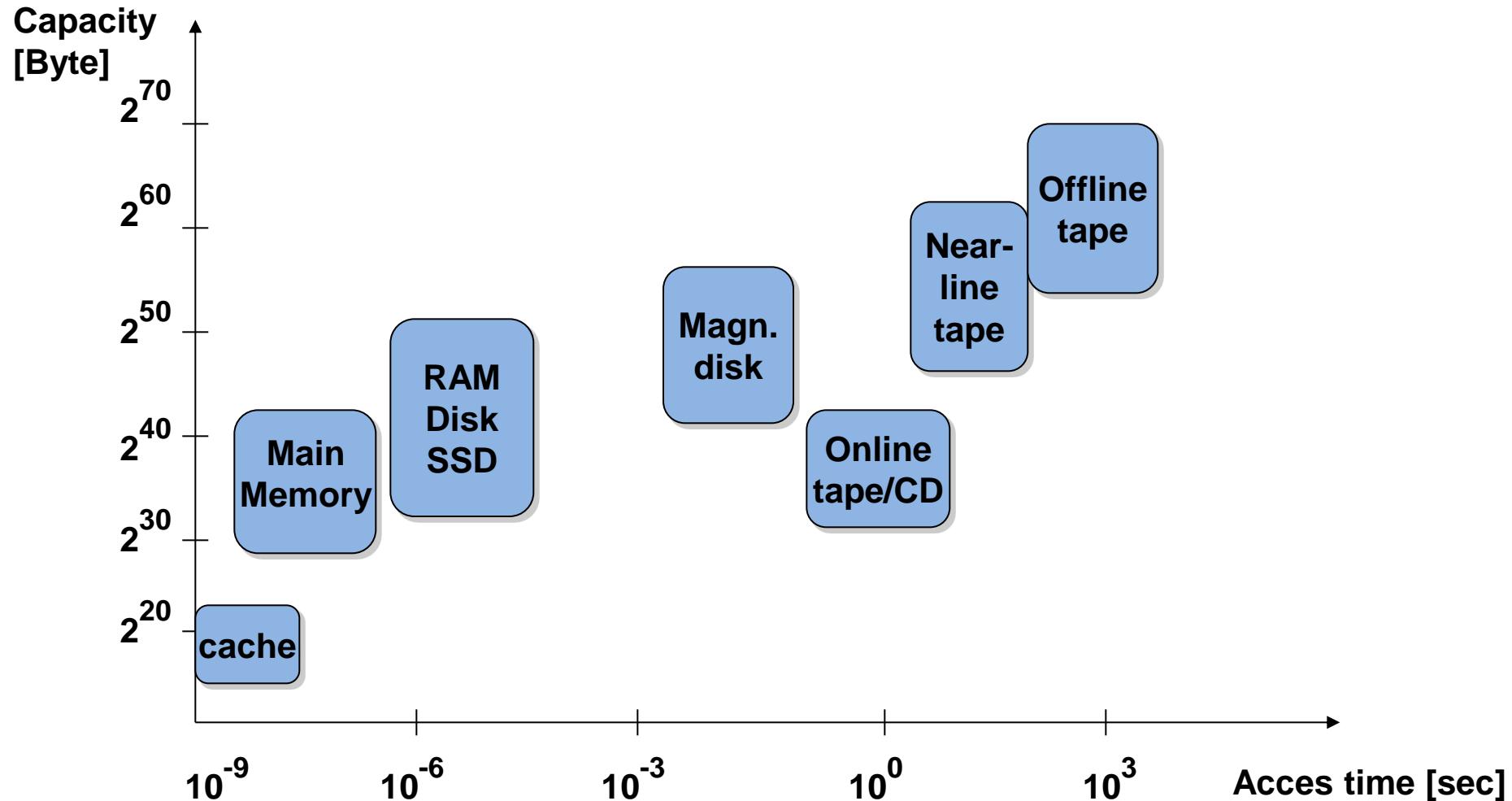
# Storage Hierarchies w.o. SSD

- RAID
- Storage hierarchy and trends
- Disk space management



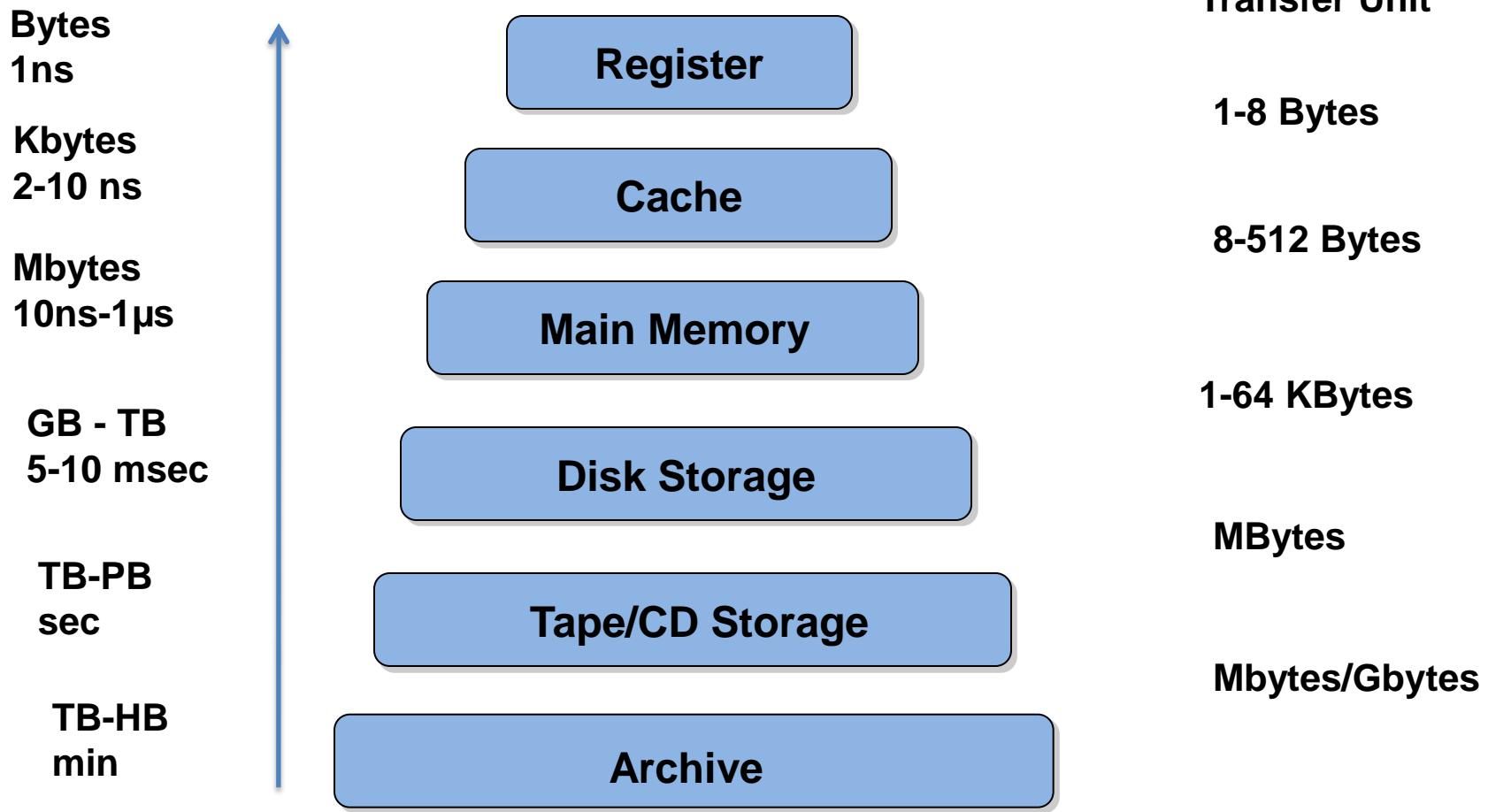
# Storage Hierarchies w. SSD

- RAID
- Storage hierarchy and trends
- Disk space management



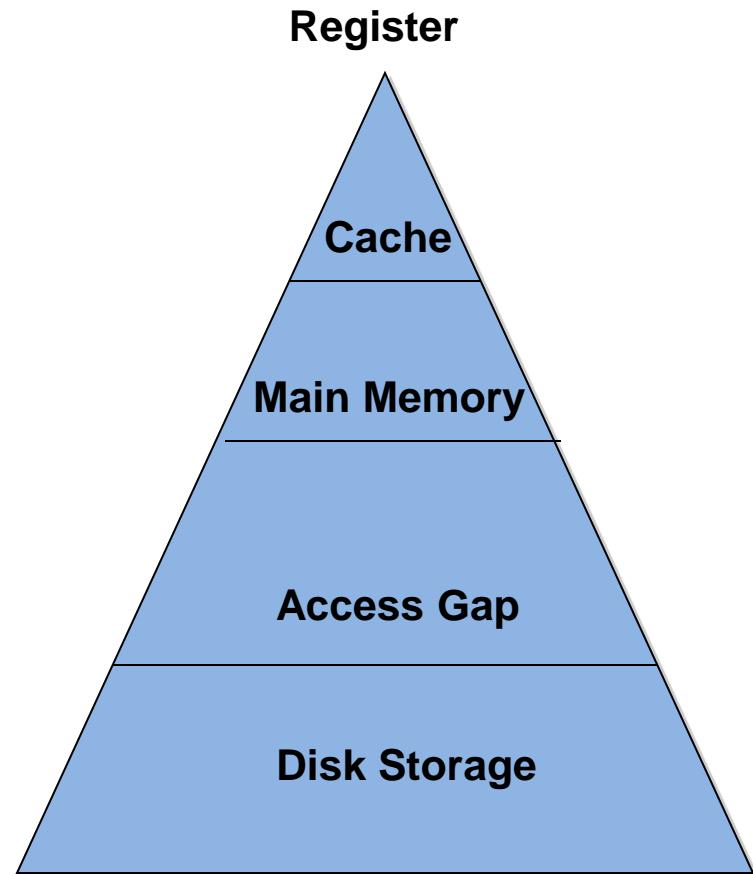
# Storage Hierarchies

- RAID
- Storage hierarchy and trends
- Disk space management



# Dealing with the Access Gap

- RAID
- Storage hierarchy and trends
- Disk space management



Typical Access Times (Granularities)  
2ns (4B)

10ns (4B)

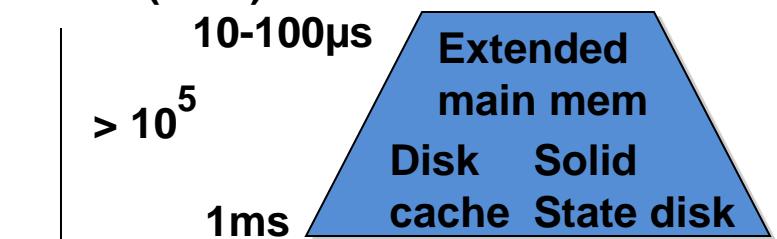
100ns (64 B)

$10-100\mu s$

$> 10^5$

1ms

10 ms (8KB)



# Relevant References RAID

- Patterson, D.A., Gibson, G., Katz, R.H.  
*A Case for Redundant Arrays of Inexpensive Disks (RAID)* ACM SIGMOD Intl. Conf. On Management of Data, 1985 (also as SIGMOD Record Vol 17, No.3)
- Chen, P.M., Lee, E.K., Gibson, G., Katz, R.H., Patterson, D.A.  
*RAID: High-Performance, Reliable Secondary Storage*  
ACM Computing Surveys, Vol. 26, No. 2, June 1994.
- Good current description: <http://en.wikipedia.org/wiki/RAID>

# RAID = Redundant Arrays of Inexpensive Disks (now Independent Disks)

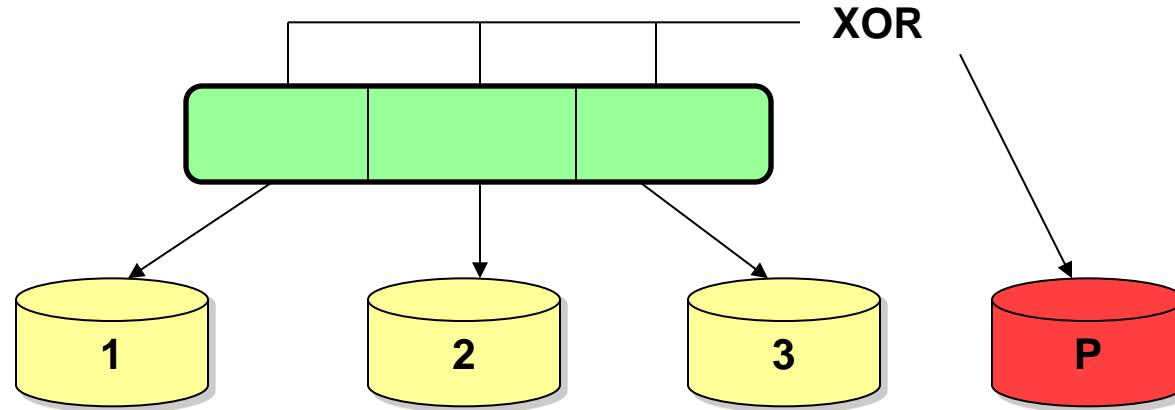
DBMS architecture revisited

RAID

Storage media and hierarchy

## Two basic ideas:

- *Data striping for performance* - data is distributed (transparently to the user) across multiple disks



- *Redundancy for reliability* - multiple (inexpensive) disks  
→ failure probability of array is sum of individual failure rates → need for redundancy

# Striping

- Multiple parallel I/O operations possible
  - *independent* I/Os → shorter waits (queuing time)
  - *multiblock* I/Os → coordinated parallel access to multiple disks → better transfer rate
- Granularity of interleaving - size of distribution units
  - *small granularity* → every I/O must access all disks
    - high transfer rate
    - 1 logical I/O at a time
    - all disks must position for each I/O
  - *large granularity* → each I/O accesses few disks
    - lower transfer rate
    - parallel I/O possible

# Reliability - MTTF

## MTTF (*Mean Time To Failure*)

- High quality disk  $(2)(10^5) - 10^6$  h or 20-100 yrs
- Standard components ~ 10 yrs

## MTTDL (*Mean Time To Data Loss*)

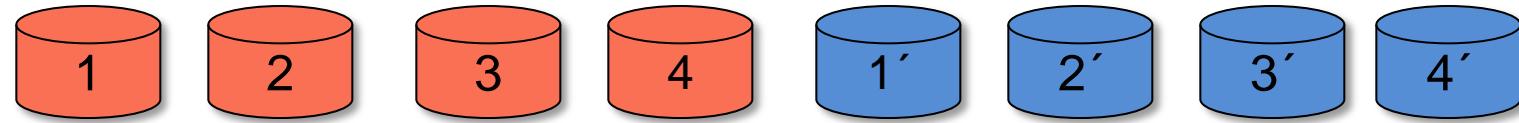
- assume independent failures
- without redundancy and  $n$  disks  $\text{MTTDL} = \text{MTTF}/n$
- *Example:*  
 $\text{MTTF} = 10 \text{ yrs.}, n = 100 \text{ disks} \rightarrow \text{MTTDL} \sim 36 \text{ days}$
- Not acceptable  $\rightarrow$  need redundancy to increase reliability

# RAID 0

- ◊ DBMS architecture revisited
- ◊ RAID
- ◊ Storage media and hierarchy

- Not truly RAID because there is no redundancy
- Only block level striping
- Improves throughput but not reliability
- Requires a controller for coordination of accesses





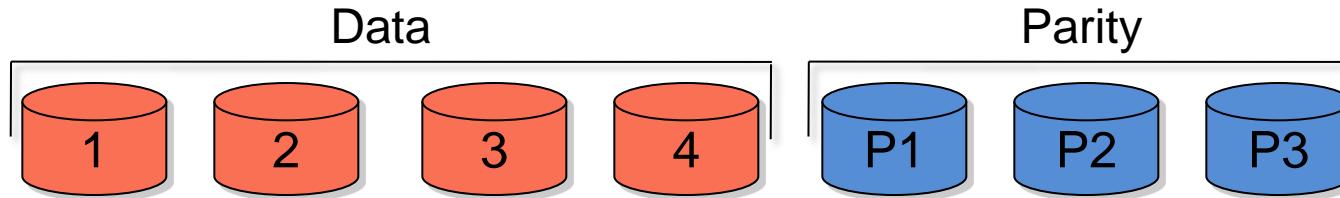
- Twice as many disks needed:  
 $Number\ redundant\ disks = Number\ data\ disks$
- Same data is *written to primary disk and mirror*
- Data is always fully redundant
- Data can be read from either copy
- Frequently used in applications where availability and transaction rate are more important than storage efficiency (e.g. OLTP)

# RAID Level 2 (Memory Style ECC)

DBMS architecture revisited

RAID

Storage media and hierarchy



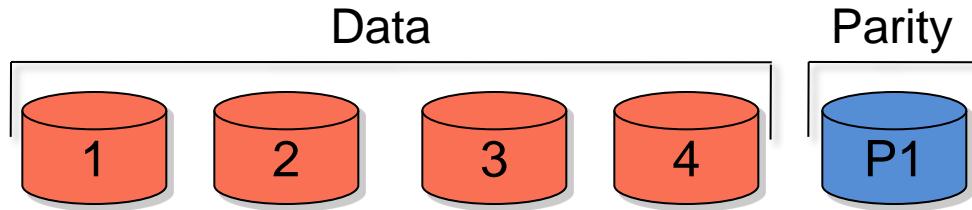
- Main memory uses Hamming codes for providing parity for distinct overlapping subsets of components
- *Number redundant disks* =  $\alpha \log$  (total disks)
- 3 extra disks needed for 4 active disks → more efficient as total number of disks increases
- Single component failure is identified by accessing the other components over which the information is distributed (only one is needed to recover)

# RAID Level 3 (Bit-Interleaved Parity)

DBMS architecture revisited

RAID

Storage media and hierarchy



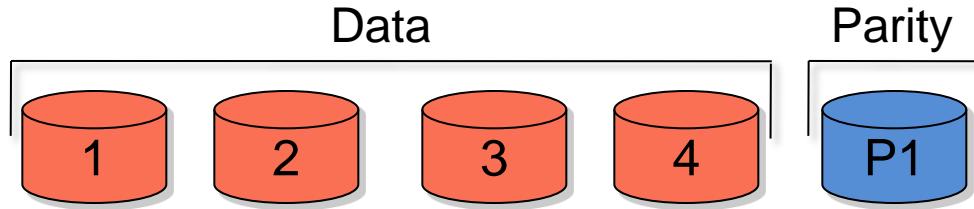
- Disk controllers can identify failed disk (simpler than identifying memory failures) → *single disk for parity*
- Data conceptually interleaved bitwise over data disks
- Single parity disk to recover from failure
- Write must access all data disks + parity disk
- Read accesses all data disks → 1 request at a time
- Good for applications requiring high bandwidth but not high I/O rates

# RAID Level 4 (Block-Interleaved Parity)

DBMS architecture revisited

RAID

Storage media and hierarchy



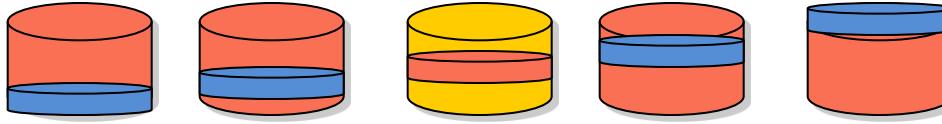
- Interleaving across disks in blocks of arbitrary size (striping unit)
- *Read*: 1 disk access if accessed unit < block size
- *Write*: update requested data blocks, compute parity, update parity disk
- Small writes use differential info (4 accesses: 1 w(new), 1 r(old data), 1r(old parity), 1w(new parity))
- Large writes use XOR over all touched disks
- Parity disk becomes bottleneck

# RAID Level 5 (Block-Interleaved Distributed Parity)

DBMS architecture revisited

RAID

Storage media and hierarchy



- Parity distributed uniformly over all disks
- Eliminates hot-spot on parity disk
- Data distributed over all disks (better read performance)
- Best small, large read, large write performance
- Small write inefficient compared to RAID 1
- Strategy for distribution of striping units important
- Left symmetric: accesses each disk once before accessing any disk twice



# RAID 6 (Block-level striping with double distributed parity)



- provides fault tolerance from two drive failures;
- array continues to operate with up to two failed drives
  - makes larger RAID groups more practical, especially for high-availability systems
  - increasingly important as large-capacity drives lengthen the time needed to recover from the failure of a single drive
  - Single-parity RAID levels are as vulnerable to data loss as a RAID 0 array until the failed drive is replaced and its data rebuilt; the larger the drive, the longer the rebuild will take.
  - Double parity gives time to rebuild the array without the data being at risk if a single additional drive fails before the rebuild is complete.



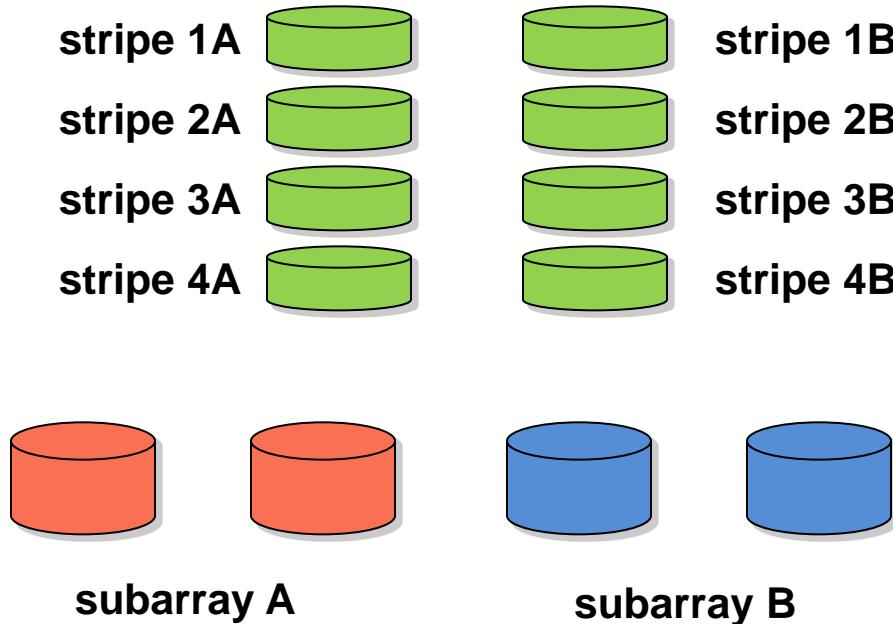
# RAID Level 10 (Hierarchical RAID)

DBMS architecture revisited

RAID

Storage media and hierarchy

- Used for high availability and very high throughput
- Consists of two layers of RAID that are combined: Level 0 (striping) on top and Level 1 (mirroring)



# RAID Level 10 (Hierarchical RAID)



- RAID 10 arrays can survive multiple disk failures
- Number of disk failures that can be tolerated is  $3^N - 1$  where  $N$  is the number of disks in a subarray
- For a 4 drive RAID 10 there are 8 possible combinations of failures  
 $\{A1\}, \{A2\}, \{B1\}, \{B2\}, \{A1,B1\}, \{A2,B2\}, \{A1,B2\}, \{A2,B1\}$
- RAID 10 uses even number of disks, *1E* is a variant for odd numbers of disks



# RAID 10 vs. RAID 01

*E.g. for ten disks:*

*RAID 01 (Mirror of Stripes):*

Divide the ten disks into **two sets of five**. Turn each set into a RAID 0 array containing five disks, then mirror the two arrays.

*RAID 10 (Stripe of Mirrors):*

Divide the ten disks into **five sets of two**. Turn each set into a RAID 1 array, then stripe across the five mirrored sets.

## Note:

Be careful with the names. *Most* of the industry use them in the way that if RAID X is applied first and then RAID Y, that is RAID XY (X/Y, X+Y). But other companies reverse the order!!! They might call a RAID XY RAID YX!

# RAID 10 vs. RAID 01

What are the differences???

**No differences in:**

- Drive requirements
- Capacity
- Storage efficiency
- Performance (nearly no difference)

→ The big difference: ***Fault tolerance!***

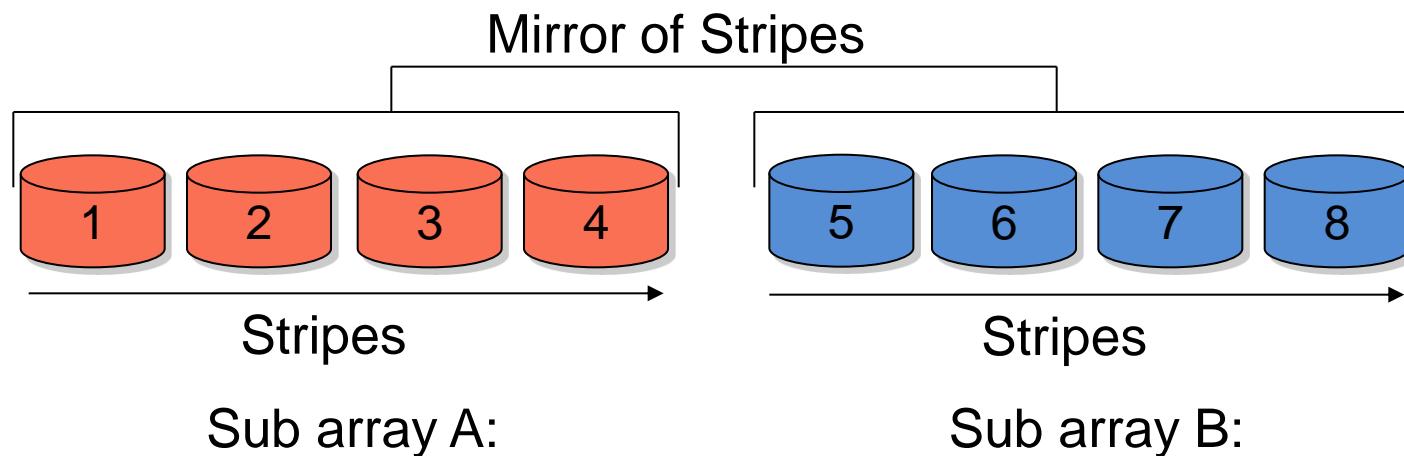
# RAID 10 vs. RAID 01

Implementation of multi level RAIDs:

- Most controllers compose a *super array* on top of *sub arrays*
- Often the *sub arrays* which form the *super array* (*also called sets*) are considered to be *single units*
- The controller only considers one of these *single units* to be *up* or *down* as a whole
- Redundancy only exists **within** a *sub array*, not **between** *sub arrays* (even if they have the same data like another *sub array*)

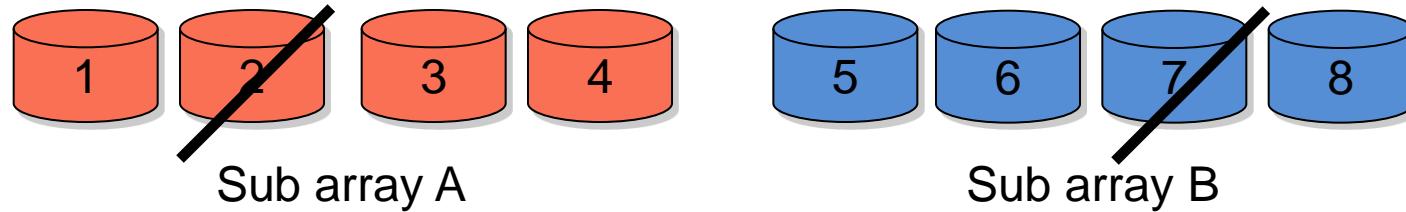
# RAID 10 vs. RAID 01

## RAID 01 with eight disks



# RAID 10 vs. RAID 01

## Fault tolerance of RAID 01



Disk 2 fails:

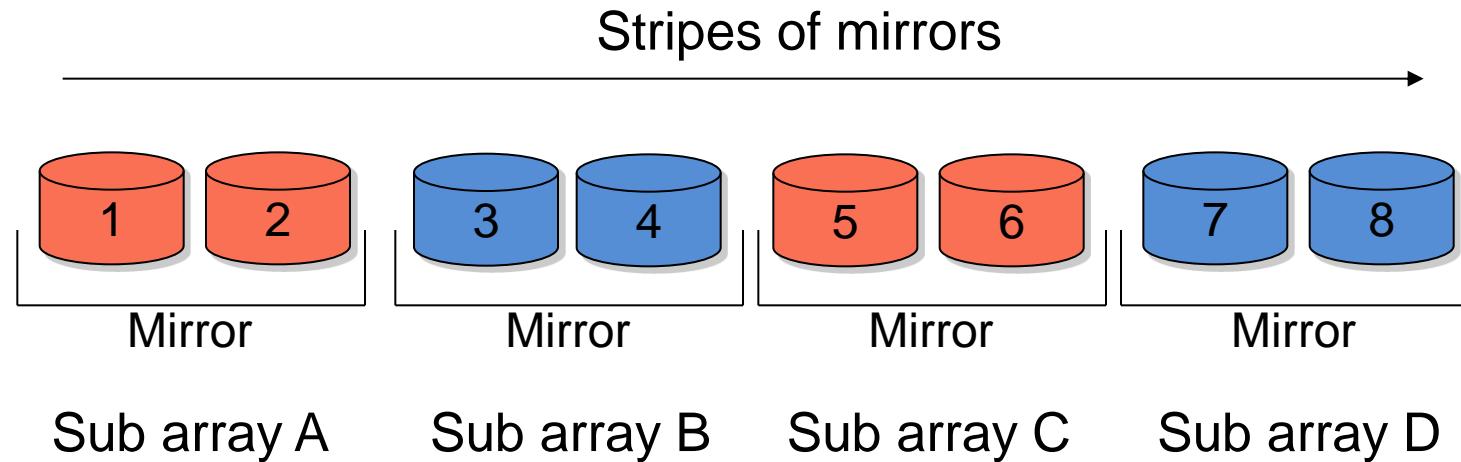
- Sub array A is down!
- Only sub array B is available

What happens if any disk of sub array B fails, e.g. disk 7?

- Sub array B is down!
- The entire array is down!

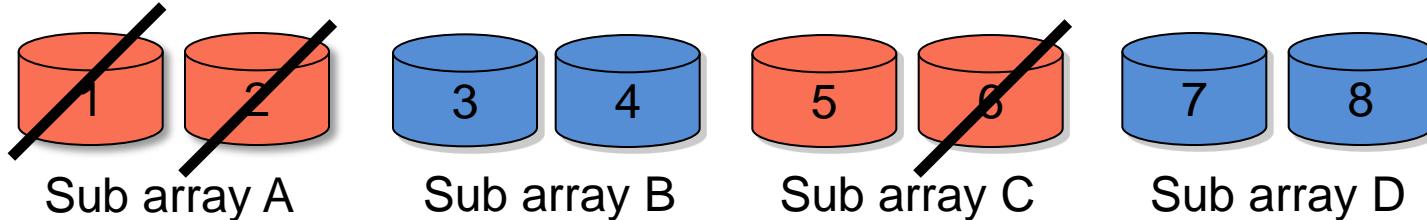
# RAID 10 vs. RAID 01

## RAID 10 with eight disks



# RAID 10 vs. RAID 01

## Fault tolerance of RAID 10



Disk 2 fails:

→ Sub array A is still available

Any other disk of another sub array fails, e.g. disk 6:

→ All sub arrays are still available

Only if *two disks of the same sub array fail*, the entire array is down (e.g., disk 1)!

# RAID 10 vs. RAID 01

## Summary

- RAID 10 is more robust than RAID 01
- Rebuilding a RAID 10 is much simpler!

*RAID 01:*

Needs to rebuild the whole stripe set of the sub array.

*RAID 10:*

Needs only to rebuild the failed disk!

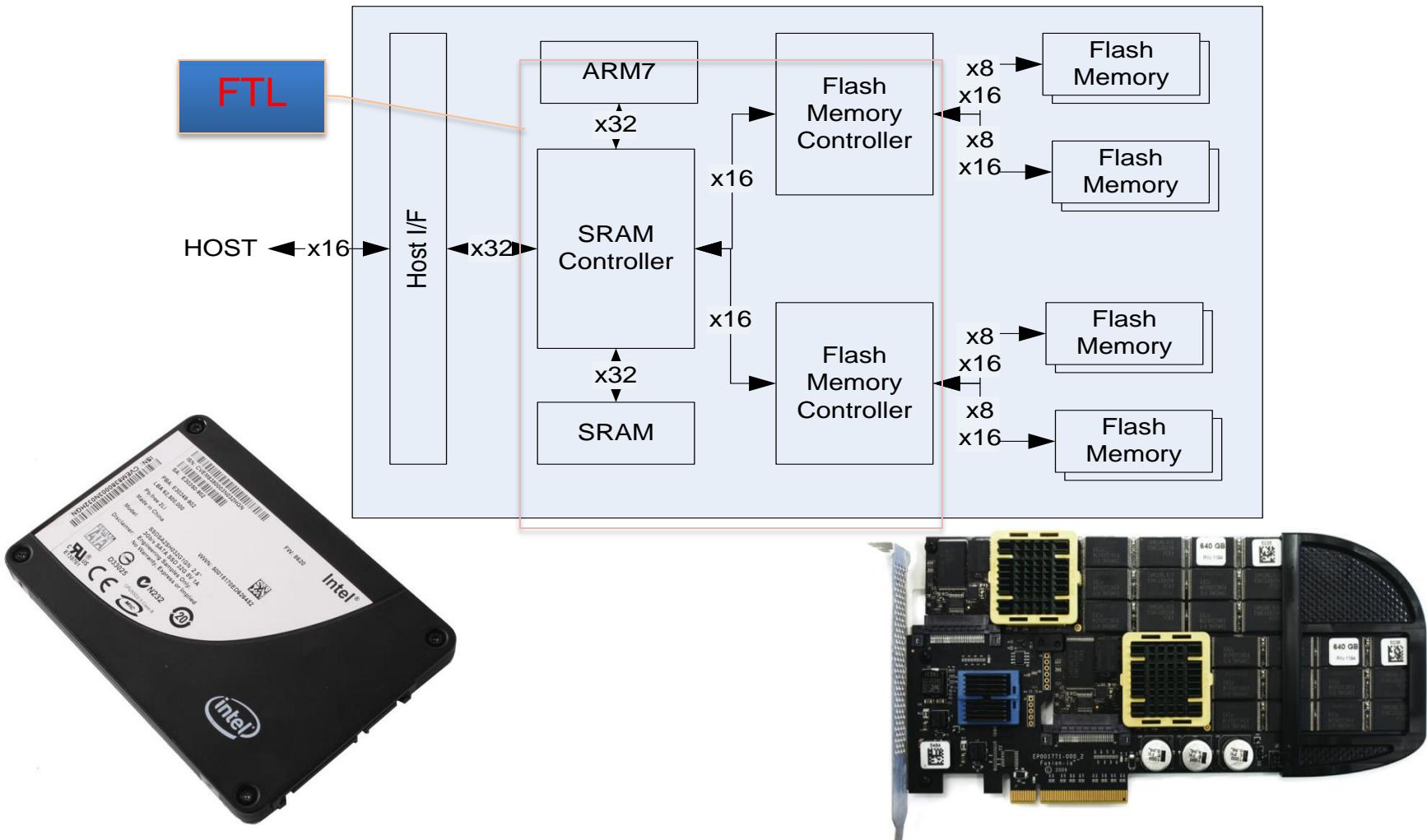
- RAID 01 could be theoretically as fault tolerant as

*RAID 10:*

If controllers running RAID 01 were smart they would use after a disk failure the corresponding disk of the other sub array.

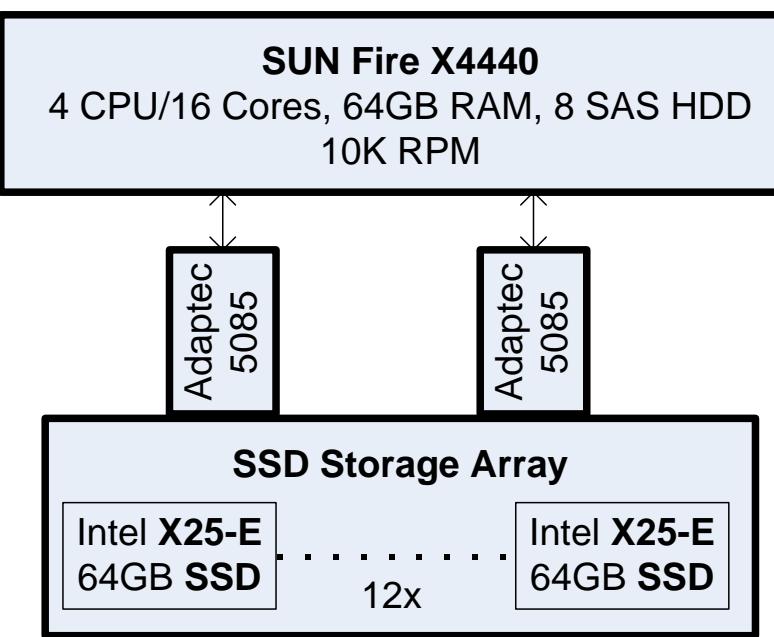
In general: **Controllers are not smart!!!!**

# Flash SSDs, X25-E, ioDriveDuo



# Specification

- Specification – Intel X25-E 64GB, SLC
- Seq. Read/Write: 250 / 170 MB/s
- Read/Write IOPS (4K): 35 000 / 3 300
- Acc. Time Read/Write (4K): 0.075/0.085 ms



- Specification: Savvio 146GB, 15k
- Seq. Read / Write: 160 MB/s
- Read/Write IOPS: 350 / 300
- Acc. Time Read/Write: 3.2 / 3.5 ms

- Comparison: ☺
  - Asymmetric read/write
    - Read/Erase/Write (read/erase blocks)
  - Very low latency
    - Position independent! (no seek)
    - Fragmentation Dependent
  - High IOPS – best for small blocks!
  - Longevity – never run SSDs at full capacity!
  - Price – high?
    - X25-E=4xSavvio 15K (dropping fast)
    - IOPS/€ ?



# And most importantly ...

- RAID controller → performance and scalability **bottlenecks**.
  - Saturate with 2 to 4 SSDs
- **Remedy:** hardware software RAID hybrids
  - Multiple RAID controllers, Few SSDs
- The SSD read/write asymmetry is amplified by RAID behavior
- Good seq. Bandwidth AND random throughput
  - Simultaneous support for multiple block sizes
- Sub-millisecond response time!
  - Frequent and **high outliers!**
- Erase block sizes influence the choice of a **stripe unit size**
  - Stripe size < erase block size (SSD specific)



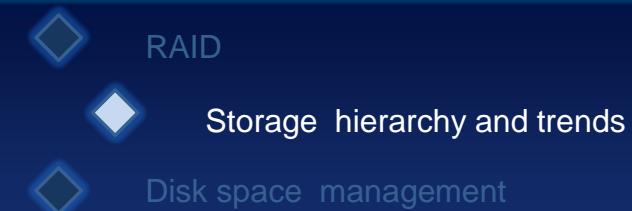
# And most importantly ...

- **Command Queuing** matters
  - Blocked I/O vs. Asynchronous Paged I/O
- Performance changes over time
  - **Fragmentation, % Used Space**



# Trends in Data Engineering

## [Gray 00, Härdler 99, Alonso 2013]



### Objective:

Examine the impact of developments in cost, capacity, and velocity of CPUs, storage and networks on DBMS design.

- Increased circuit density leads to higher memory chip capacity and faster CPUs.
  - Larger memories require larger addresses:
    - 1970: 16 bit addresses
    - 2000: 36 to 40 bit physical addresses  
*36 bit required to address 64GB memory*
    - 2007: 64 bit addresses
- 
- A horizontal timeline arrow pointing to the right. Three vertical tick marks are placed along the arrow, each aligned with a text label below it. The first tick mark is labeled '16bit' above the arrow and '1970' below the arrow. The second tick mark is labeled '36-40bit' above the arrow and '2000' below the arrow. The third tick mark is labeled '64bit' above the arrow and '2007' below the arrow.
- Logical addresses provided by common systems:
    - 64 bit (SPARC, Itanium, MIPS, PowerPC) to 96 bit (AS400)
    - 1 extra bit required every 18 months



# Rules of thumb - Joy's law

- 35% p.a. increase in CPU Mips rates from 1970 to 1990
- RISC technology speeds up growth rate:

## Joy's law

$$\text{SunMips}(\text{year}) = s \times 2^{(\text{year}-1984)} \text{ for years from 1984-2000}$$

- Doubling capacity yearly proved too optimistic
- Effectively capacity doubled *approx. every 18 months*
- ~ 60% effective annual growth in Mips rate

# Rules of thumb - Moore's law (RAM capacity)

RAID

Storage hierarchy and trends

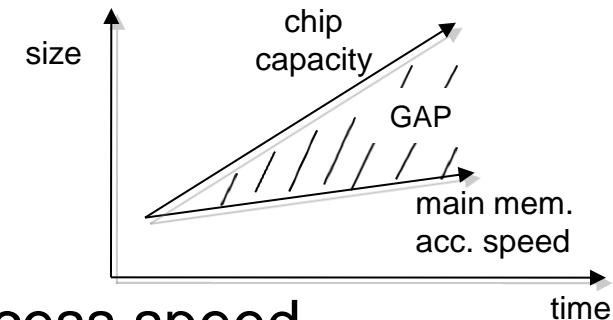
Disk space management

- Moore's law originally postulated for RAM, has been generalized for µprocessors and disk storage capacity

## Moore's law (RAM capacity)

$$\text{MemChipCap}(y) = m \times 4^{(y-1970)/3} \text{ Kb/Chip for [1970...2000]}$$

- Chip capacity increased every 3 years by a factor of 4
- Main memory access speed improved only approx. 7%/year
- Imbalance between capacity and access speed
- Must compensate with fast caches



# Rules of thumb - disk capacity

- RAID
- Storage hierarchy and trends
- Disk space management

## Hoagland's law for magnetic areal density (1985):

$$\text{MagArealDensity}(y) = d \times 10^{(y-1970)/10} \text{ Mb/in}^2 \text{ for [1970-2000]}$$

Hoagland's law proved too pessimistic (factor 10 in 10 years or ~30%/yr)

- Disk capacity is better modeled by Moore's law adapted for disk capacity (over the years from 1985-2000)

$$\text{Disk Capacity}(y) = c \times 4^{(y-1985)/3} \text{ Mb/in}^2 \text{ for [1985-2000]}$$

Corresponds to observed factor of ~1000 in 15 yrs.



# Amdahl's Law

**Amdahl's law states that ....**

***each CPU instruction per second requires one byte of main memory***

- Ratio of CPU instr./sec to bytes of main memory =  $\alpha$
- $\alpha = 1$  according to Amdahl's law
- Up to 2005 computers had  $\alpha$  between 3 and 10
- What is happening with multicore and main memory DBs?



# Amdahl's Law - speedup

## Amdahl's law

$$S = \frac{1}{(1-f) + \frac{f}{k}}$$

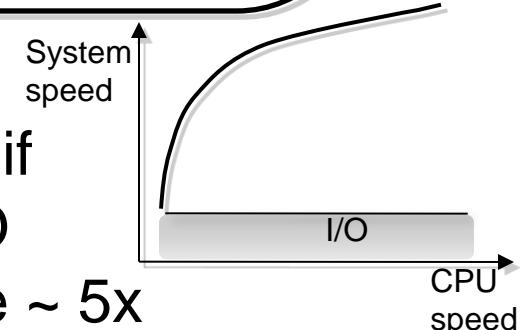
where  $S$  effective speedup

$f$  fraction of work in faster mode

$k$  speedup while in faster mode

Consequence:

If CPU is 10x faster but I/O is bottleneck, if system spends just 10% of time doing I/O (slower mode), maximum speedup will be  $\sim 5x$



# Consequence of Amdahl's law

- Reduce ratio of fast to slow operation
  - Flash memory/SSD vs. rotating magnetic disk
- Reduce fraction of operation in slow mode
  - Trend towards main memory databases, but ...
    - You still need persistence
    - You must be able to recover from failure (restore MM from persistent medium)

- RAID
- Storage hierarchy and trends
- Disk space management

# World's first Harddrive

## IBM 350:

- 50 Discs
- 1 Head
- 5 Mbyte
- 1 Ton



(Source: IBM)



# World's first Harddrive

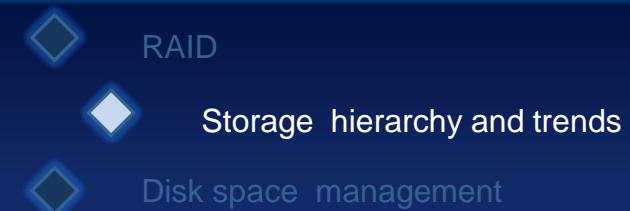
## IBM 350:

- 50 Discs
- 1 Head
- 5 Mbyte
- 1 Ton

(Source: IBM)



# Reference values for disks [Gray 2000] and Seagate 06

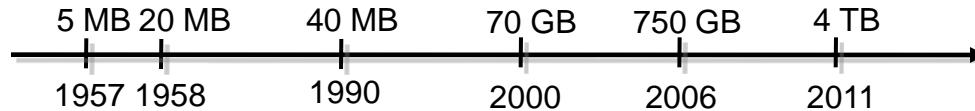


- **Magnetic areal density** 1985: 20 Mpsi  
2000: 35 Gpsi  
2006: 421 Gpsi (Seagate)

[http://www.wwpi.com/index.php?option=com\\_content&task=view&id=1392&Itemid=39](http://www.wwpi.com/index.php?option=com_content&task=view&id=1392&Itemid=39)

- **Capacity** 2000: 70 GB 2007: 1.2 TB  
2006: 750 G 2011: 4 TB

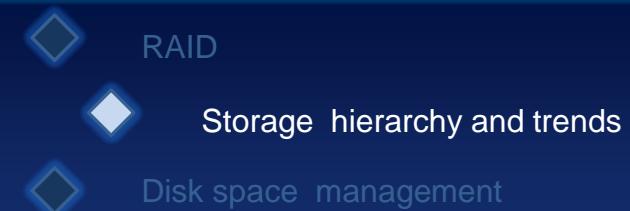
<http://www.seagate.com/cda/products/discsales/guide/>



- **Access time**  
2000: 10 msec (120 kaps - Kbyte access/sec)  
2006: 3.5 msec @ 15 000 rpm



# Reference values for disks [Gray 2000] and Seagate 06



- **Transfer rate**

1975: 25 MBps (~ 20 maps Mbyte access/sec)

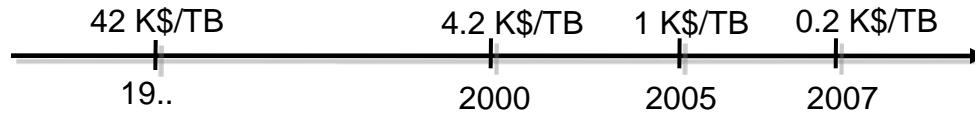
2006: 320 MBps

- **Scan time**

2000: 45 min

2005: 2 hr

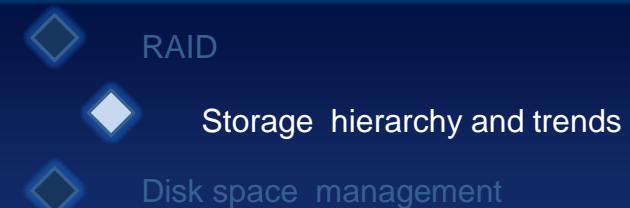
- **Cost** 42 K\$/TB (large system), 15 K\$/TB (low performance IDE drives packaged) in 2000 dropped dramatically by factor 10 (2007: 0.2 K\$/TB (low performance IDE drives packaged))



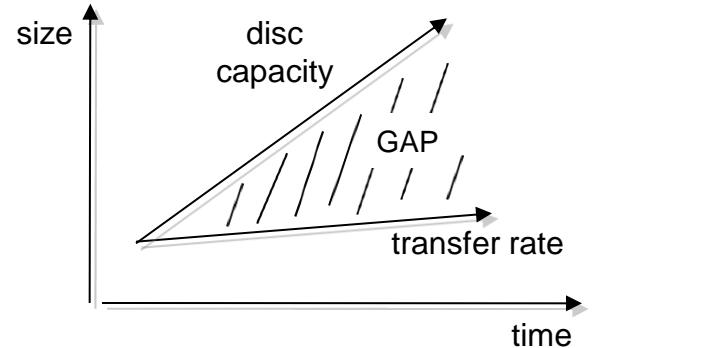
- 2005: same form factor stores 0.5 TB with transfer rate of 75 MBps resulting in 2 hr scan time and cost ~ 1 k\$/TB



# Effects of current trends



- Gap in access time for MM to acc. time for disk  $\sim 10^5$
- Disk capacity improved 1000x over 15 yrs
- Transfer rate improved only 40x over same 15 yrs
- Ratio between disk capacity and disk access per second increasing > 10x per decade
- Ratio capacity/bandwidth increasing > 10x per decade

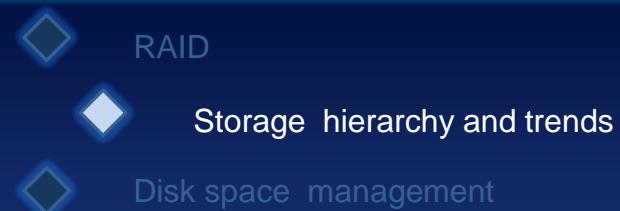


## Effects:

- 1) disk accesses become more precious
- 2) data becomes cooler with time (more time between accesses)



# How to deal with trends



- Use large transfer units (e.g. whole track)
  - page size grew from 2 KB to 8 KB, expected to grow to 64 KB
  - Oracle recommends a default page size of 8192 Bytes (8KB)
    - fewer spanning records, less waste at end of page, faster full table scans

Burleson et al. *Oracle Space Management Handbook*  
Rampant Press, 2003

- Favor sequential accesses
  - one random access requires  $t_{seek} + 0.5 t_{rot} + t_{transf}$
  - sequential access proportional to  $t_{transf}$
  - whole table scan vs. index use (break even between 1-2%)

# How to deal with Trends (cont.)



- Reevaluate RAID strategy
  - mirroring doubles read capacity and costs 1 extra write
  - RAID5 uses up to 4 disk accesses/write, improves read bandwidth when data requests go to different physical disks
- Use different mountpoints (different disks for different physical accesses belonging to same logical access, e.g. data, index, recovery data)



# How to deal with storage trends



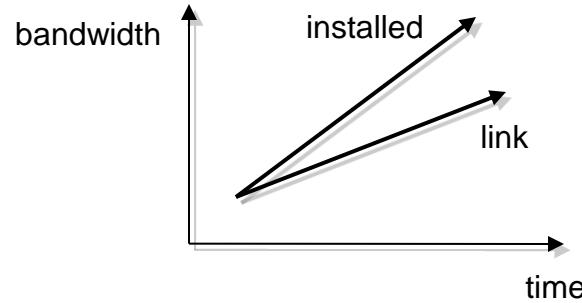
- Flash memory is getting cheap, esp. if we consider cost per IOP
- Implications for DBMS:
  - Parallelism within DBMS becomes less important
  - Selective reads become more attractive
  - Append operations are more efficient → scanning index
  - R/W asymmetry becomes more important (minimize writes)
  - Recovery time better than with magnetic disks
- What about new storage technologies (e.g. PCM – phase change memory)?



# Trends in Network Bandwidth and their Effects



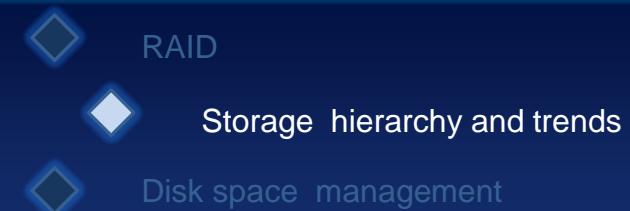
- Installed bandwidth is doubling every 9 months
- Link bandwidth improves 4x every three years



- Latency is constant (e.g. delay via satellite bounded by speed of light, 60ms round trip time within US or Europe)
- Bulk-transfer of data is cheap, minimize latency



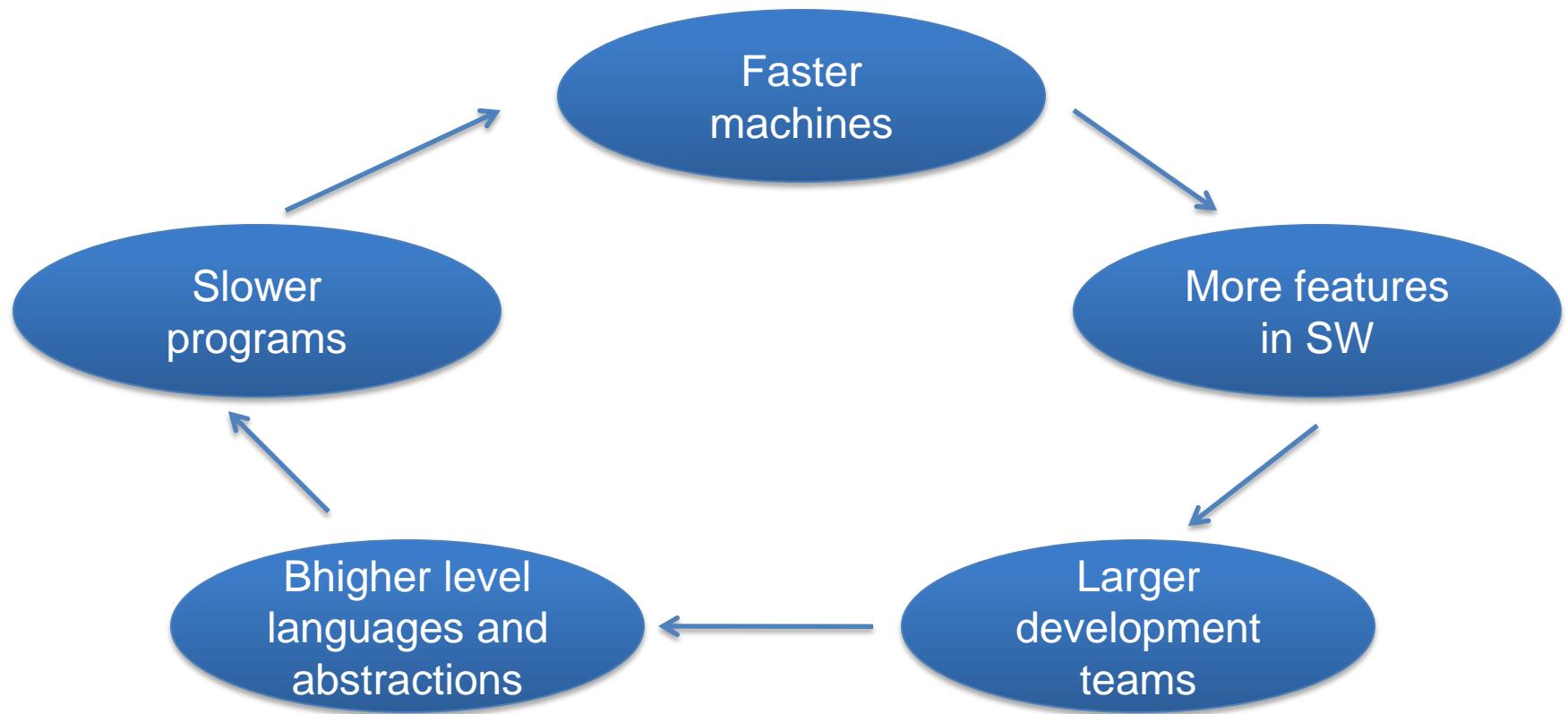
# Trends in Network Bandwidth and their Effects(cont.)



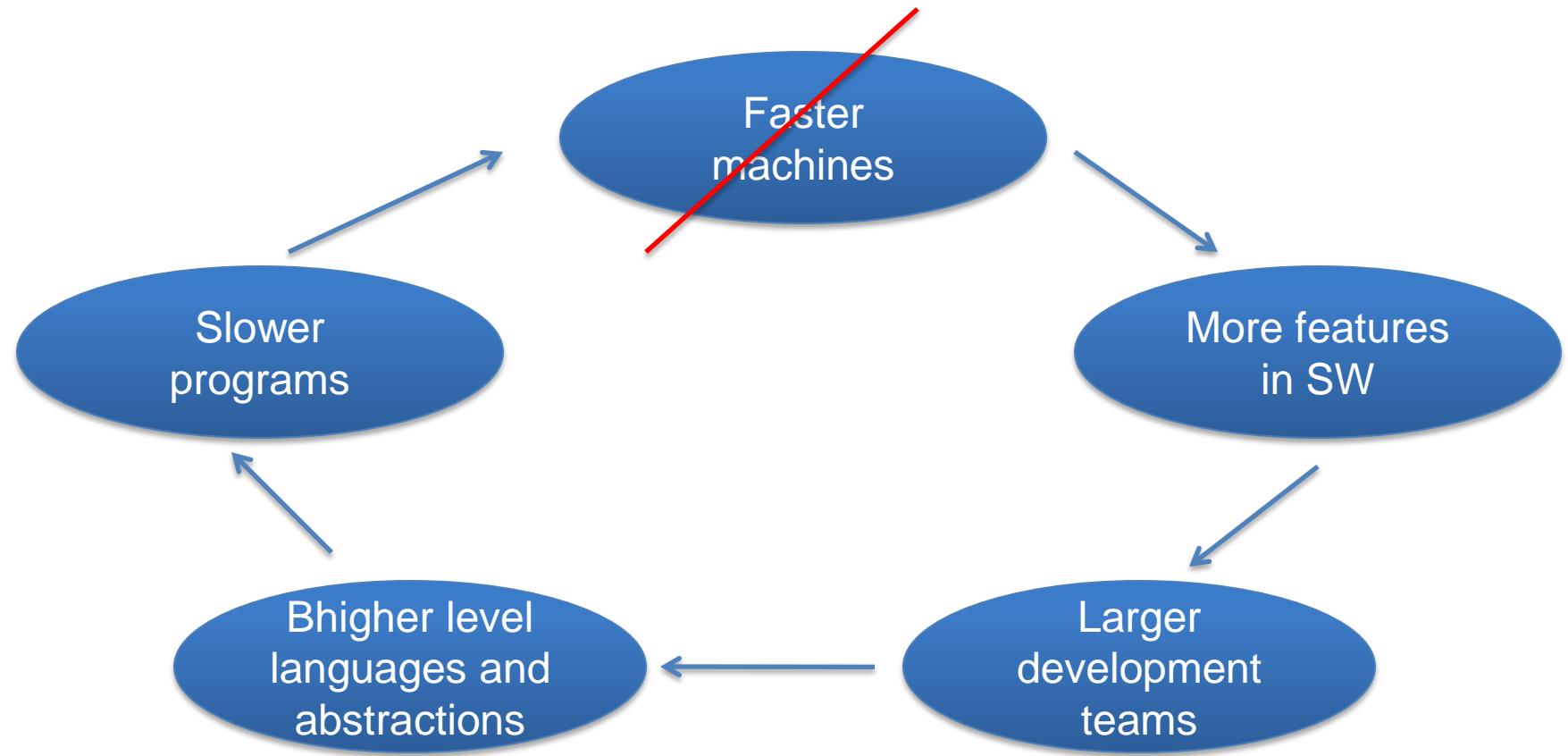
- Trend towards distribution and caching
  - service location problems
  - cache consistency problems (pull vs. push)
- Favor streaming algorithms in distributed systems
  - *streaming joins* (e.g. sort merge join)
  - use of *streamable indexes* (e.g. bit map indexes)
  - transport of bulk files with streamable, offset-based *indexes*
- What about storage in the cloud?



# The virtuous cycle up to 2005 (Jim Larus)



# The virtuous cycle up to 2005 (Jim Larus)



# Trends - clock

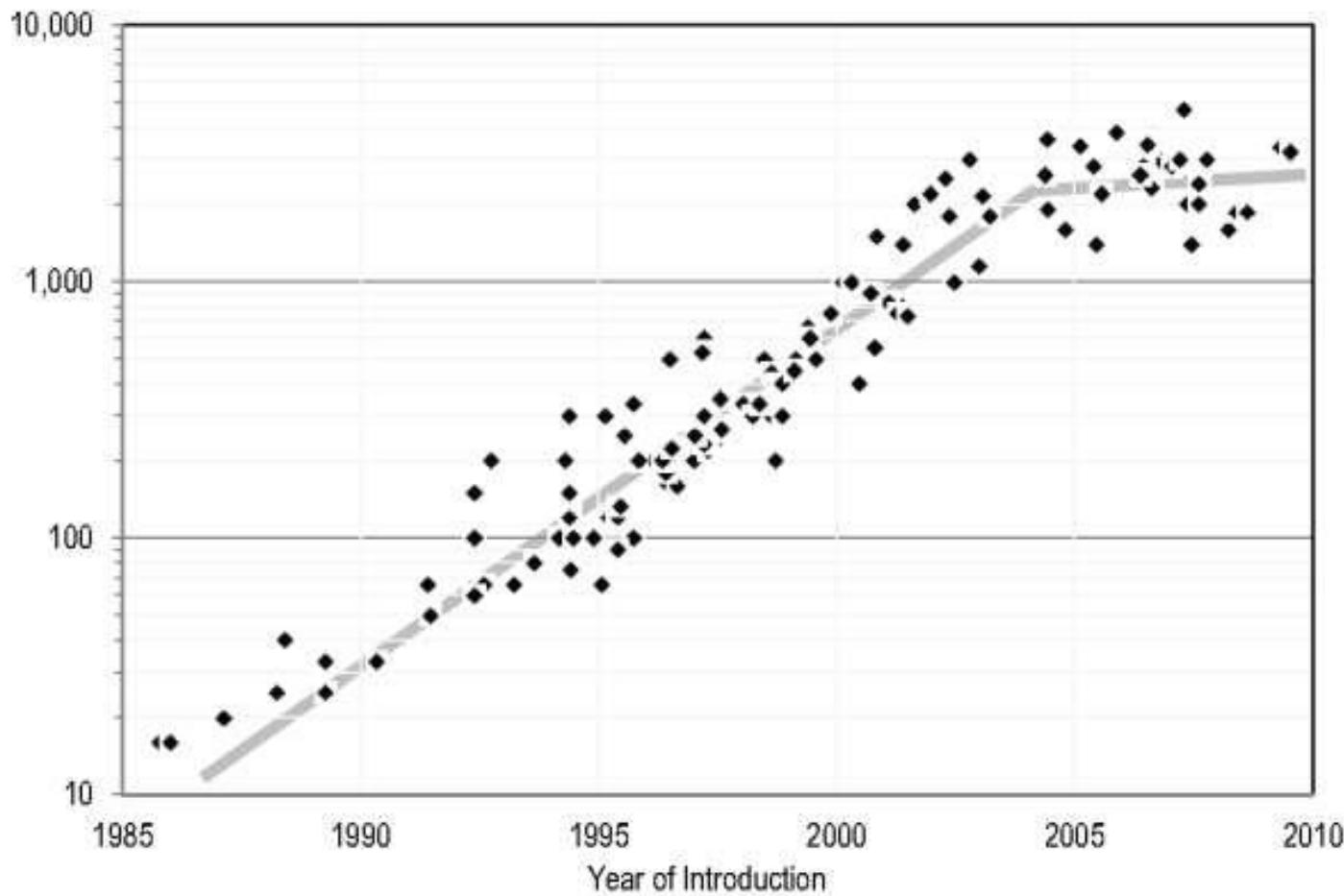


FIGURE A.3 Microprocessor clock frequency (MHz) over time (1985-2010).



# Trends – single thread performance

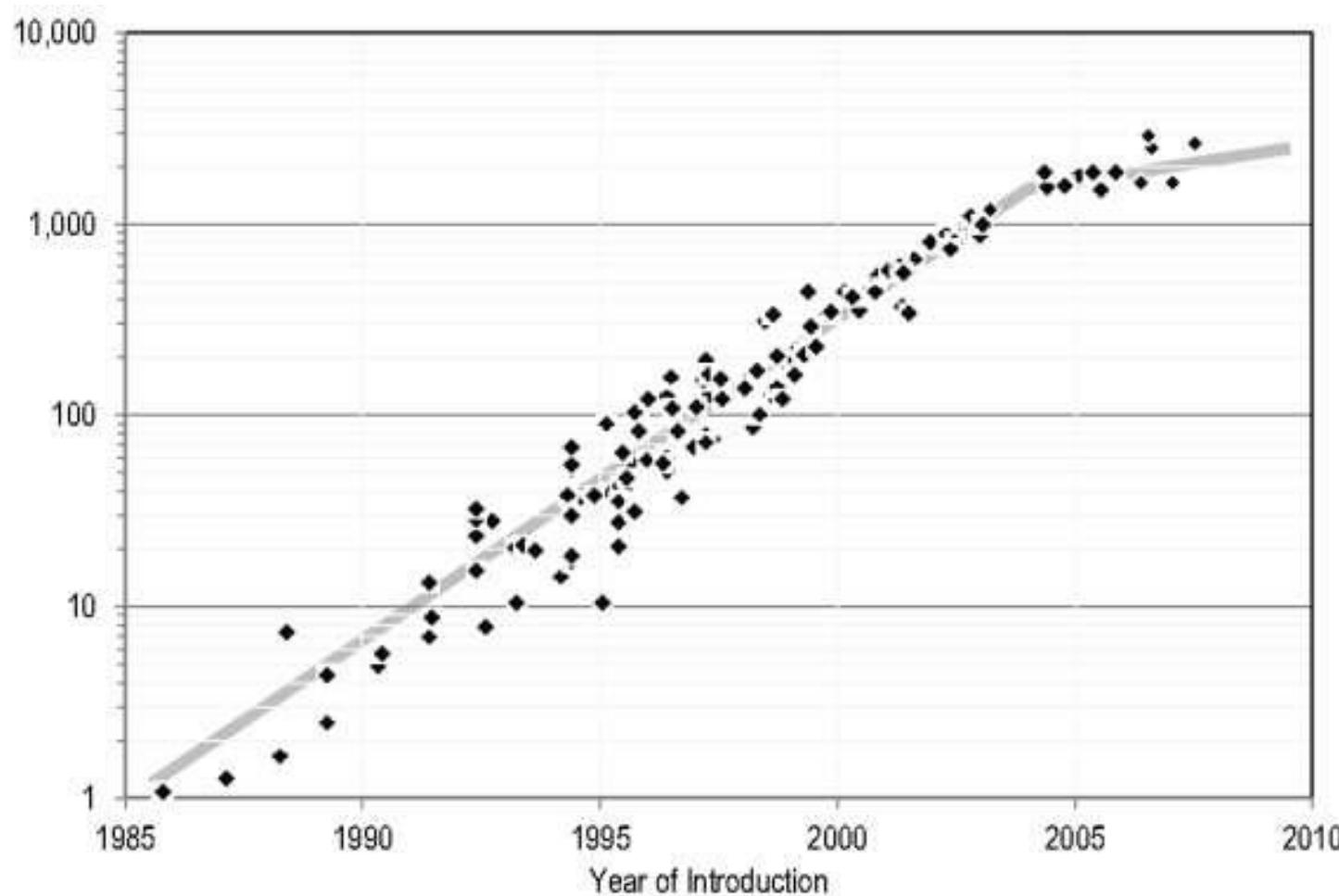


FIGURE A.1 Integer application performance (SPECint2000) over time (1985–2010).

# Trends – power dissipation (cause)

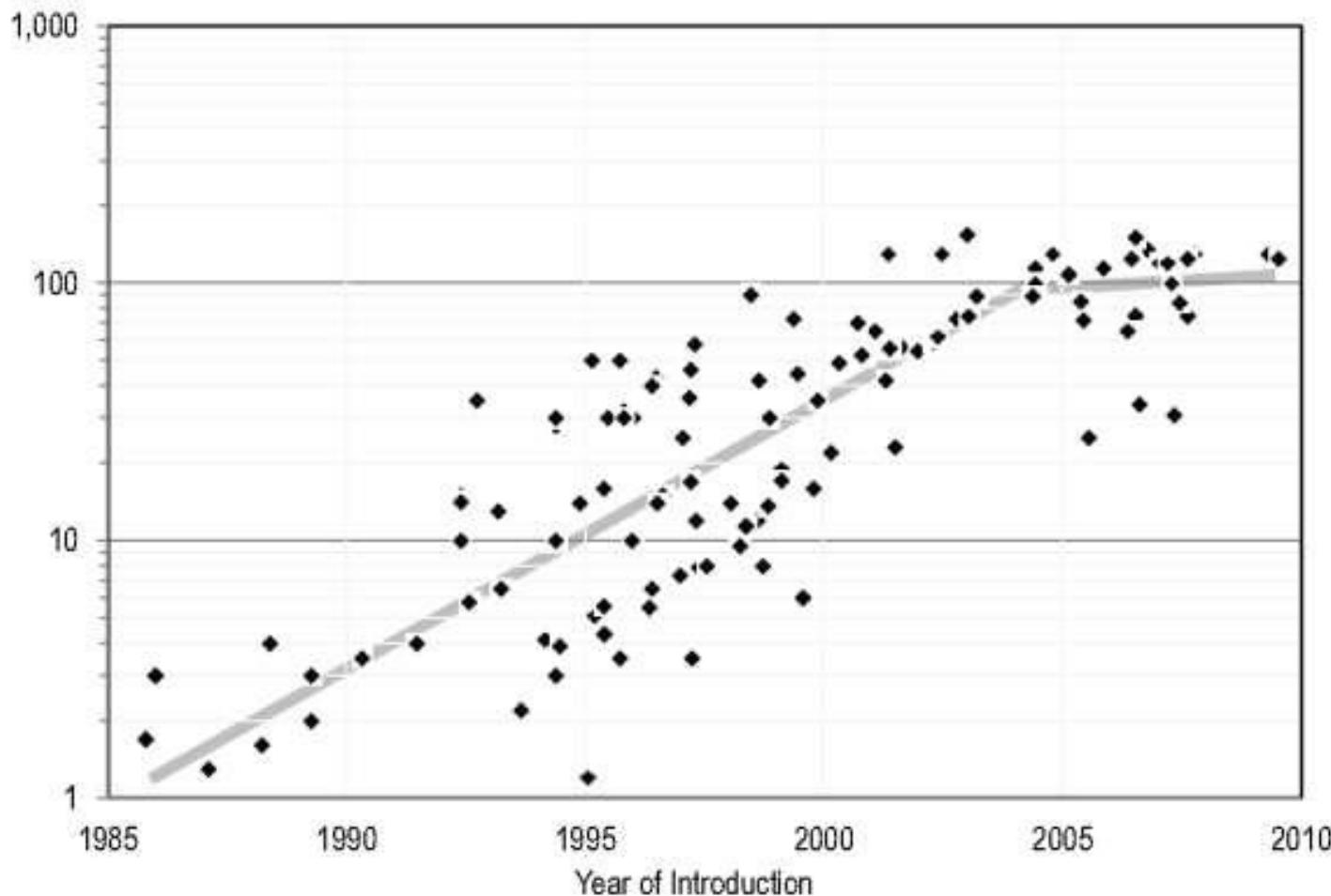


FIGURE A.4 Microprocessor power dissipation (watts) over time (1985-2010).

# Consequences of hitting the wall

Limits in power dissipation

Limits in speedup

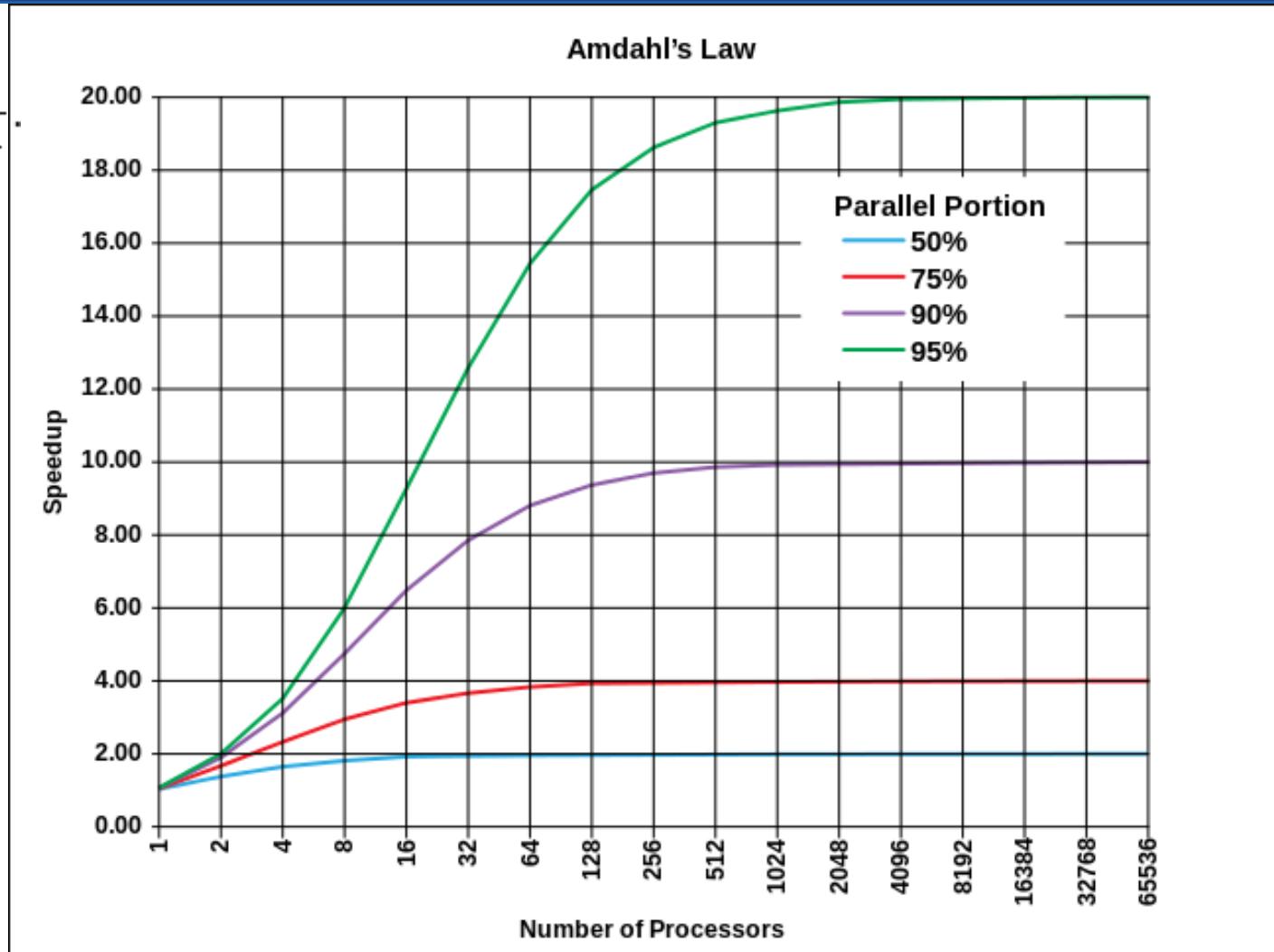
Limits in CPU performance

→ Clustering and multicore → parallelism

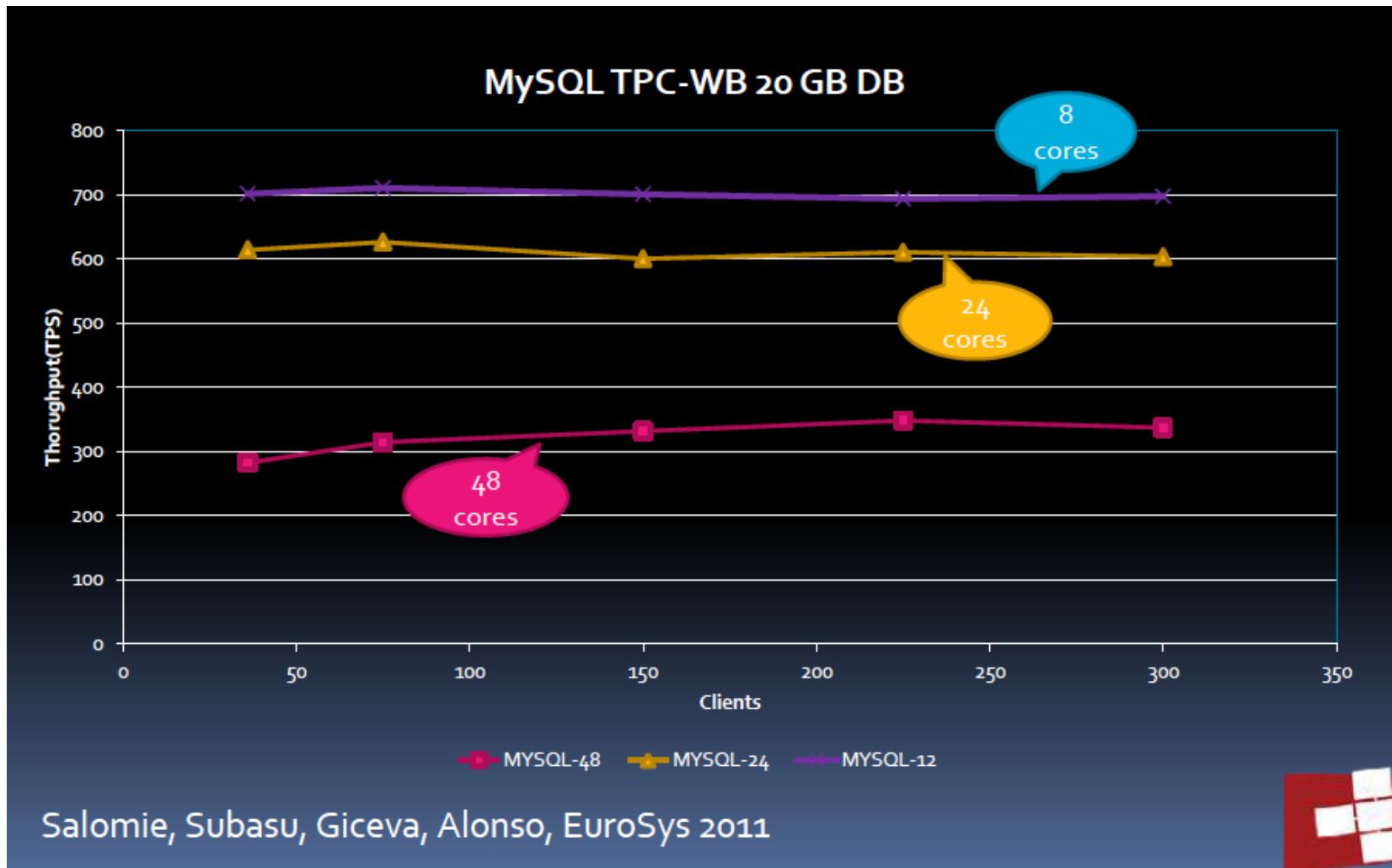


# Amdahl's law is not your friend - parallelism (Alonso ICDE 2013)

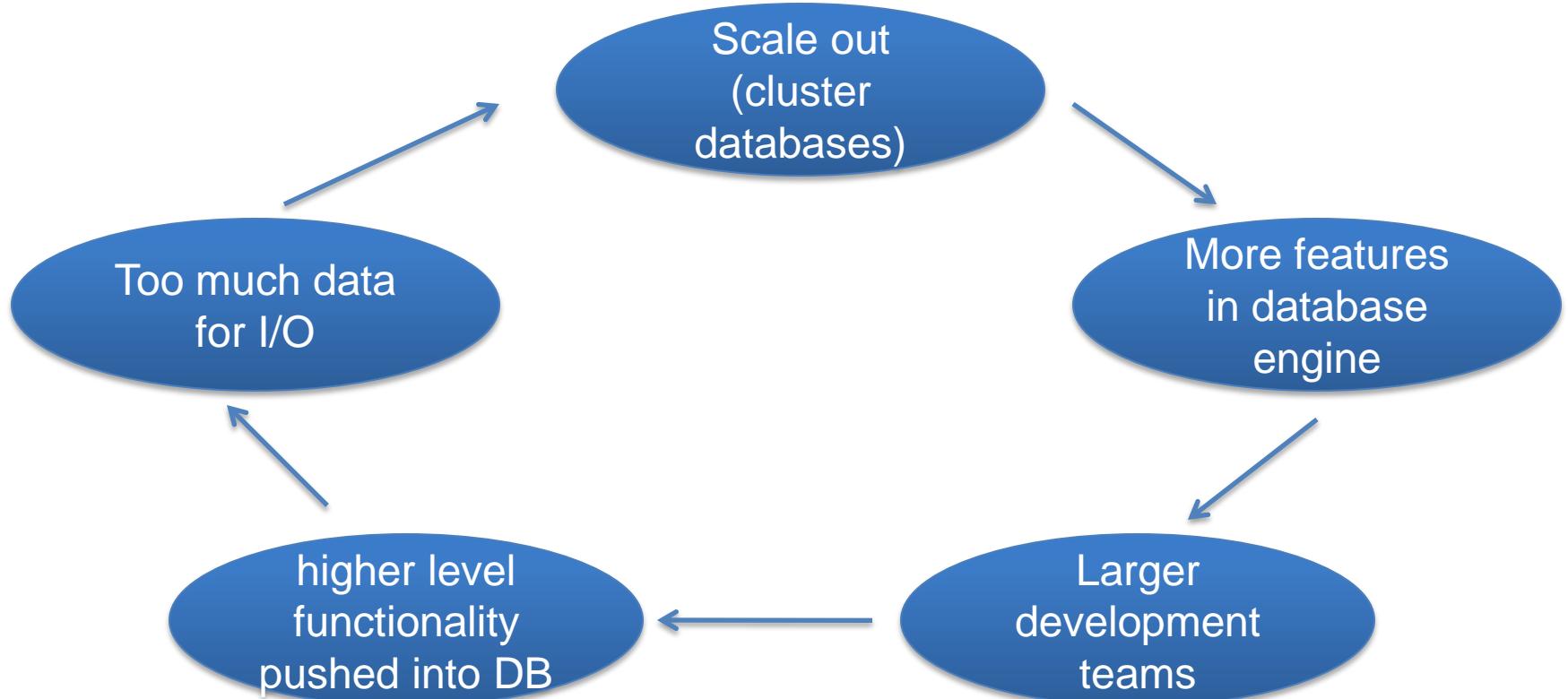
$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}}.$$



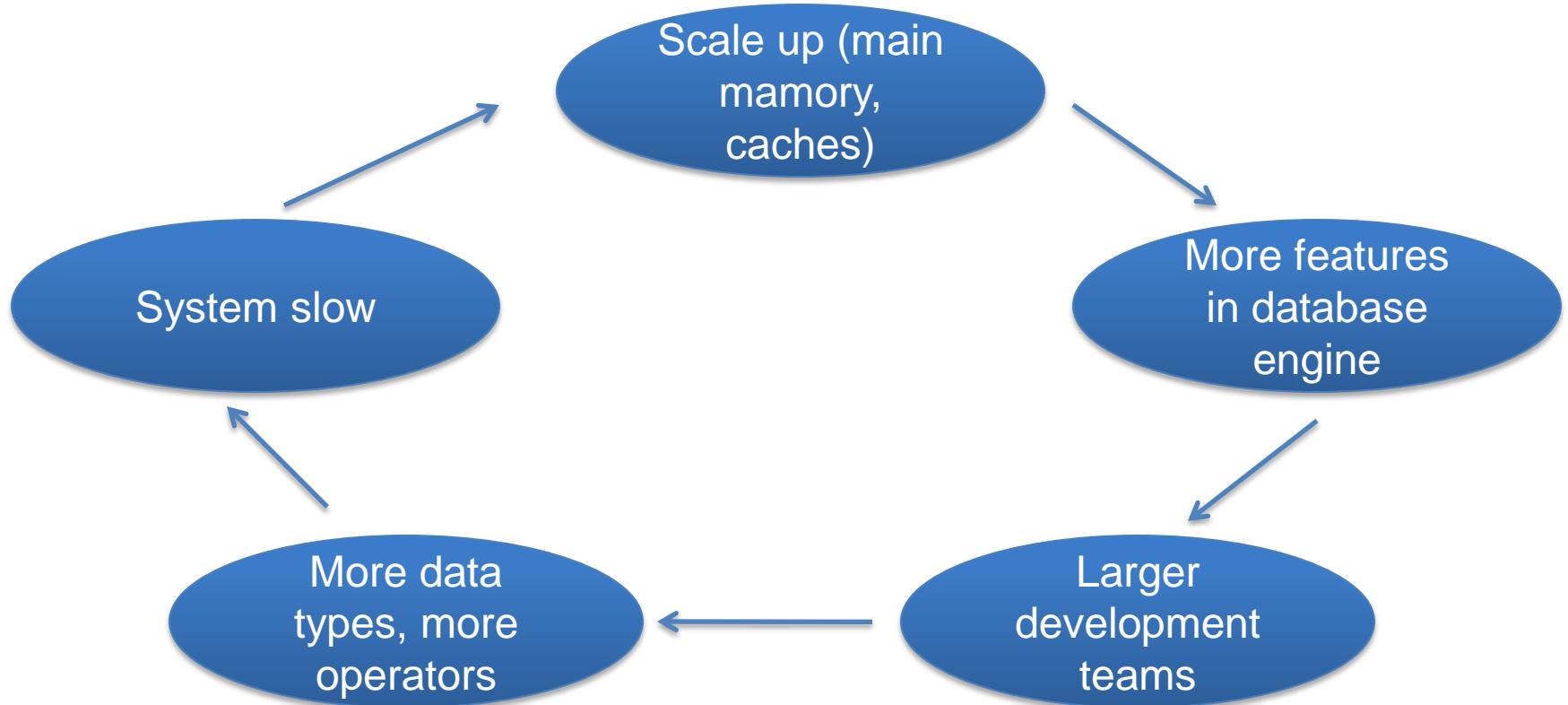
# Many systems can't use multicore properly



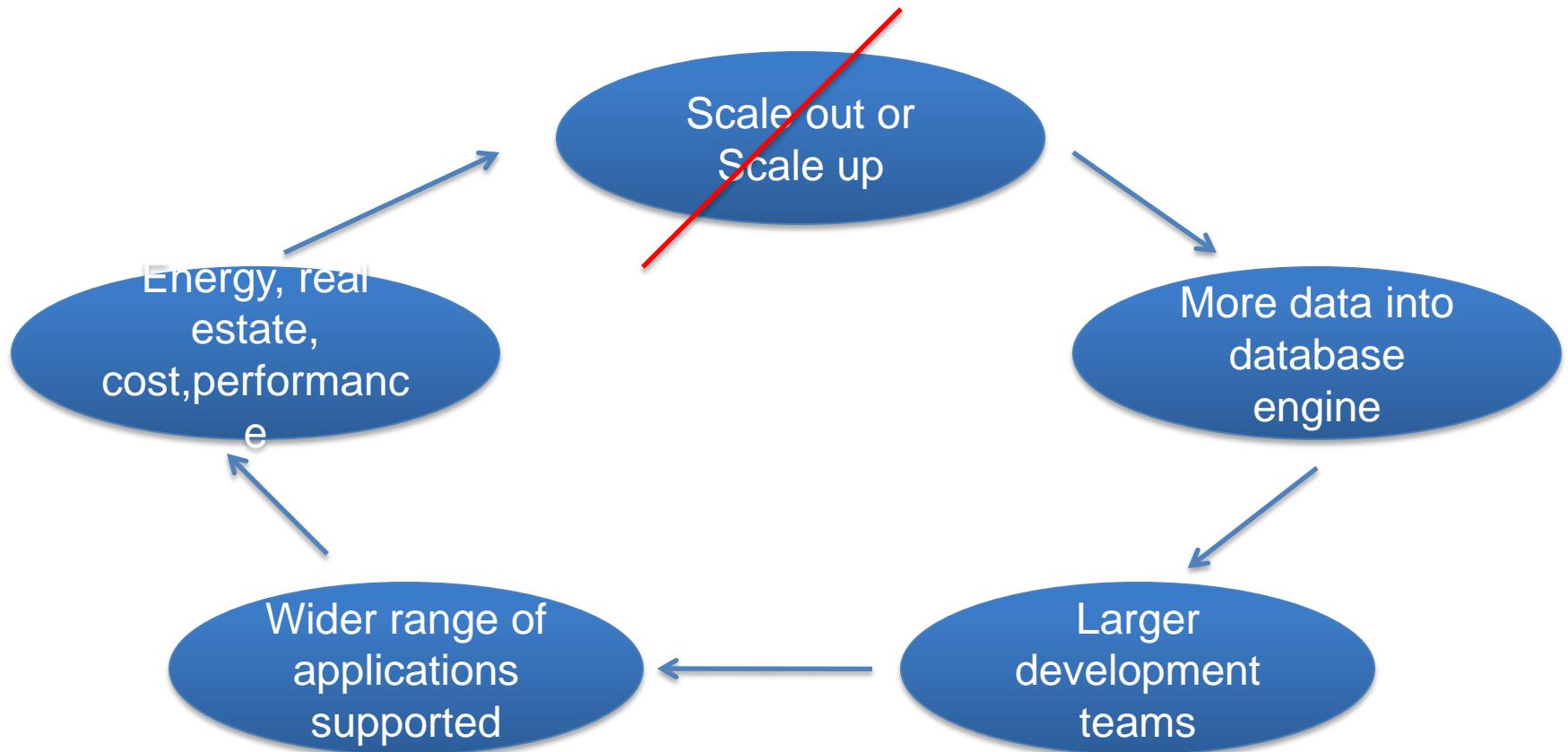
# The virtuous cycle for databases – scale out (Alonso)



# The virtuous cycle for databases – scale up (Alonso)



# We also get stuck! (Alonso)



# Consequences

- Must think out of the box
  - New hardware support (e.g. FPGA)
  - Main memory databases
    - Map reduce workloads (analysis)
      - Microsoft: median job size < 14 GB
      - Yahoo: median job size < 12.8 GB
      - Facebook: 90% of jobs < 100GB
    - Fit in main memory
  - Single node with more memory is more efficient than cluster (back to scale up vs. scale out)



# Disk Space Management



File interface  
read block i  
write block k

Addressing units: blocks, files

Auxiliary structures: free-space info., extent tables, file directories

Addressing units: tracks, cylinders, channels

Device interface  
channel programs

Physical Storage Mapping

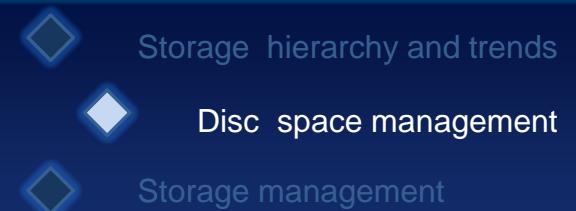
Devices

## Tasks of the disk-space manager:

- management of external storage devices
- hiding of device-specific properties
- mapping of physical blocks to external devices
- control the data transfer to and from the buffer



# The File Abstraction



- Convenience of splitting a DB into disjunct portions
- Each such disjunct portion is a file
  - selective activation/deactivation of files
  - dynamic file definition supports modular DB growth
  - temporary files can be deleted after they have been used
  - files can be allocated to different devices with varying characteristics
  - shorter addresses within a file
- File system implements files and presents common abstraction to higher level layers
- Device characteristics are hidden by the file abstraction



# The File System: OS vs. DBMS-specific



- Operating systems provide a file system that could be used as the basis for the DBMS
- Most DBMSs implement their own file management
  - portability issues - no reliance on OS-specific features (Ramakrishnan) vs. reuse of OS's basic disk access method (Härder)
  - optimized (contiguous) space allocation
  - need to span multiple devices with single file
  - exploitation of page reference patterns - DBMS can exploit page reference patterns better than OS (limited and more predictable operations, data access vs. general purpose programming)
  - better prefetching strategies

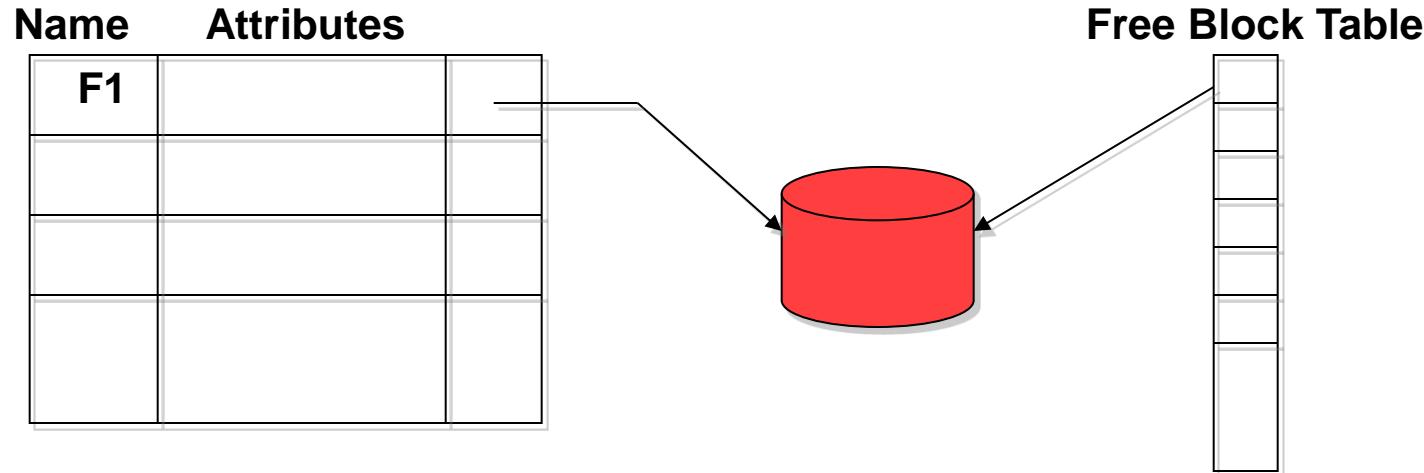


# File Systems

Storage hierarchy and trends

Disc space management

Storage management



- File descriptors describe file in catalogue through attributes: *name, ownerID, access control list, size, extents, date of creation, last access, last backup, ...*
- Free block table - typically maintained as (hierarchical) bitmap to facilitate fast identification and allocation of contiguous disk space



# Block Allocation Strategies

- Three approaches:
  - *static*
  - *dynamic*
  - *(dynamic) extent-based*

## Static allocation

- |  |  |
|--|--|
| <ul style="list-style-type: none"><li>- simple, allows efficient allocation of contiguous storage space</li><li>- easy address calculation</li></ul> | <ul style="list-style-type: none"><li>- reservation of the whole space at creation time</li><li>- overflow requires unloading, redefinition of file size and reloading</li></ul> |
|--|--|



# Dynamic Block Allocation

- Blocks are allocated one at a time, first time a block is written a free block is located and allocated

## Dynamic allocation

- |   |   |
|---|---|
| <ul style="list-style-type: none"><li>- extreme flexibility</li><li>- dynamic extension of file</li></ul> | <ul style="list-style-type: none"><li>- large block tables</li><li>- space fragmentation</li><li>- difficult to maintain clustering</li></ul> |
|---|---|

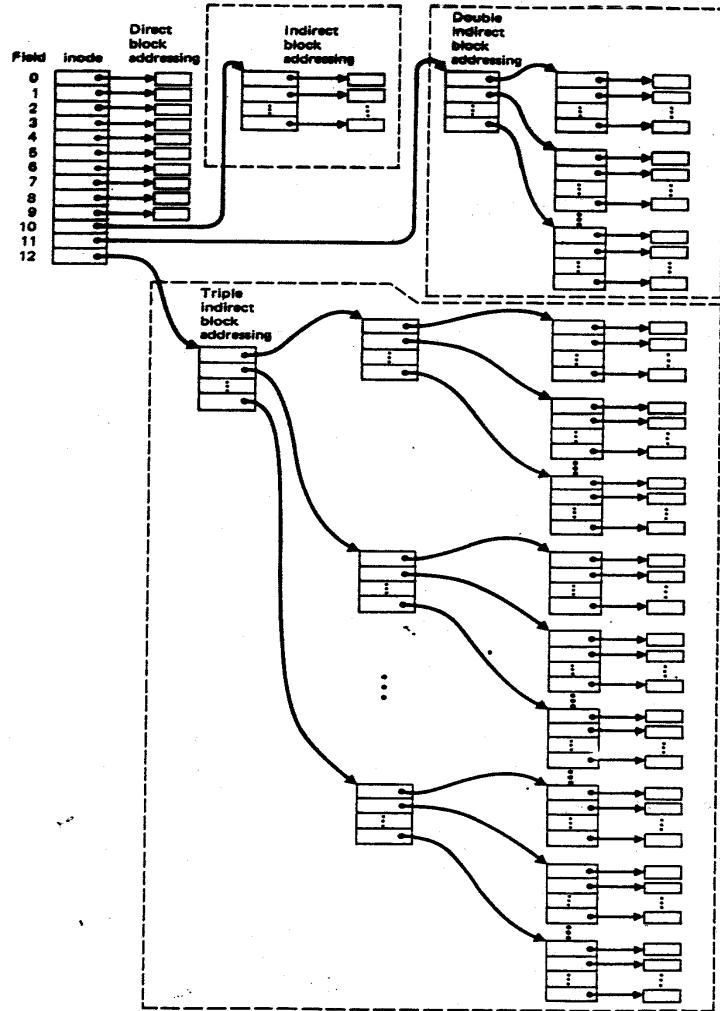
- Extreme case: UNIX file system
  - multiple levels of indirection: inode, indirect block addressing, double indirect b. a., triple indirect b.a.
  - optimized for small files

# UNIX file system

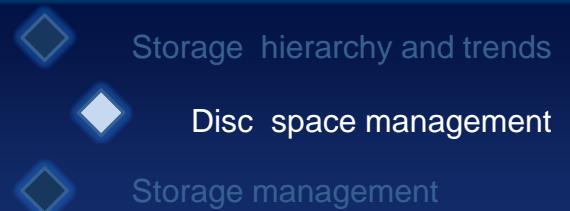
- Storage hierarchy and trends
- Disc space management
- Storage management

multiple levels of indirection:  
inode, indirect block addressing,  
double indirect b. a., triple indirect

block allocation  
optimized for small files



# Extent Allocation



- Extents represent a compromise between the flexibility of dynamic allocation and the benefits of contiguous static allocation
- Extent is a set of contiguous pages

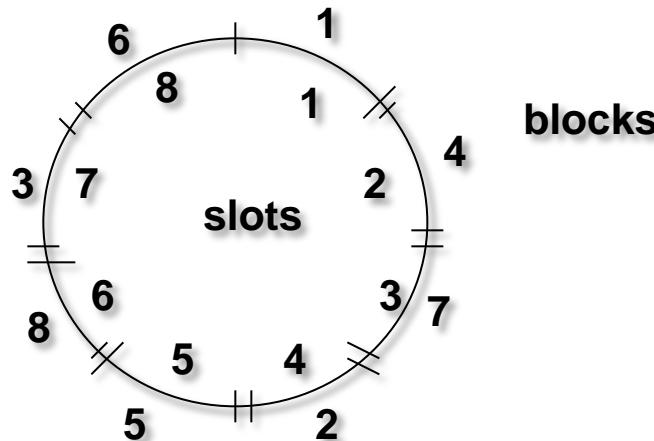


# Mapping of blocks to slots

## (Härder/Rahm)



- Blocks of a file are mapped to slots on disk
- Delays occur after reading one block (channel/bus interrupts, selection of the next I/O)
- If blocks assigned to slots strictly sequentially, delays cause loss of next read (and a whole rotation)
- Staggering of blocks when mapping to slots



$$SN_i = (SN_{i-1} + l - 1) \bmod(m) + 1$$

**$m$  = number of blocks/slots**

**$l$  = offset**

**$1 < l < m$**

**$m$  should not be divisible by  $l$**



# Disk space structure in INFORMIX



- Two distinct types of disk space possible:
  - *cooked file space* (UNIX manages physical disk I/O)
  - *raw disk space* (Universal Server manages physical disk I/O)
- Different physical units used to manage disk space:
  - chunk, page, blobpage, sbpage, extent
- Logical units overlayed on physical units
  - dbspace, blobspace, sbspace, extspace, database, table, tblspace
- Special purpose disk storage structures
  - logical log, physical log, reserved pages



# Chunks



- A ***chunk*** is the largest unit for allocating disk space in INFORMIX Universal Server
  - *primary chunks*
  - *mirrored chunks* (automatically activated if primary fails)
- Chunks generally limited to 2 GB (some systems 4GB)
- Maximum of 2048 chunks per US system
- *Raw disk space* - contiguous space on a UNIX block-type file with character-special interface
- Raw disk guarantees
  - better performance because of DMA (direct memory access)
  - more reliable, guaranteed writing to disk (cooked file space may write only to a UNIX buffer)



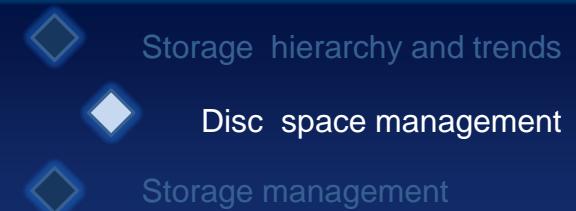
# INFORMIX US Pages



- **Page** is physical unit of disk storage used by US to read from and to write to disk
- Page size is HW-determined, typically 2 or 4 KB
- Chunks contain pages. Pages are fully contained within a chunk
- *Blobpages* are the units of disk space allocated for storing simple large objects (BLOBs). Blobpage consists of  $n$  pages
- *Sbpages* are similar to Blobpages for smart large objects
- Size of Blobpages/Sbpages should accomodate 1 large object (design for the most common case)



# INFORMIX US Extents (similar in Oracle)



- Physical unit of storage used to allocate initial and subsequent storage space
- Typically there are two types of extent associated with a table
  - *initial-extent* (size = number of KB allocated initially to a table)
  - *next-extent* (size = number of KB allocated in each increment)
- Extents are always fully contained in a chunk
- If US can't find enough contiguous space (next-extent size) in the present chunk it searches for it in another chunk



# Oracle Physical Database Storage Units

Storage hierarchy and trends  
Disc space management  
Storage management

- *The physical database storage units*, data files, are associated with *table spaces* according to the logical structure of the database.
- *Table spaces* may be created to separate different categories of data.
- *Table spaces* are divided into smaller logical divisions called segments, which are divided further into extents and data blocks.



# Pages and Segments

-  Disc space management
-  Storage management
-  Buffer management

DB buffer interface  
get page i  
free page j

Addressing units: pages, segments

Auxiliary structures: page tables, block tables  
segment definitions, etc.

Addressing units: blocks, files

Page Mapping Structures

File interface

- Additional level of abstraction above blocks and files
- Introduction of pages and segments
  - segments with special properties can be designed
  - allows in-place and shadow updates
  - segments can be used as larger locking/recovery granules

-  Disc space management
-  Storage management
-  Buffer management

# Segments (Härder/Rahm)

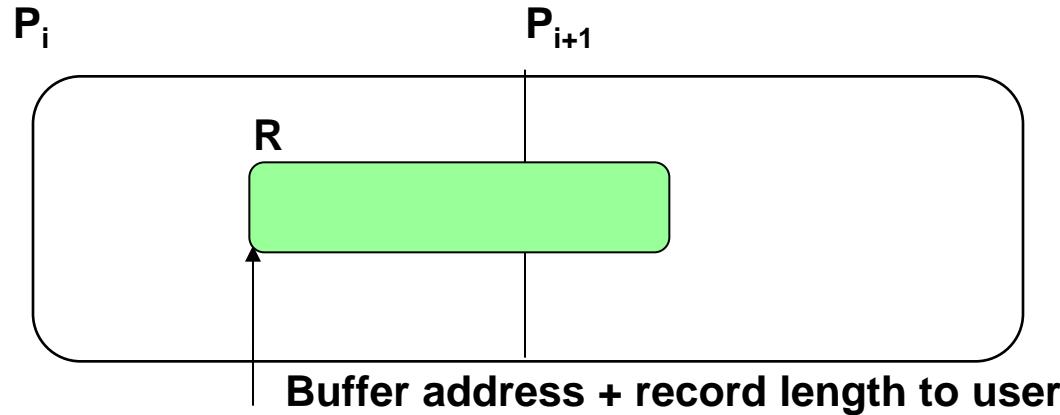
Properties	Types	Type 1	Type 2	Type 3	Type 4	Type 5
Use		public	private	private	private	private
Durability		Perm.	Perm.	Perm.	Perm.	Temp. in transaction
Opening/Closing		Automatic by system		Explicit by user		
Recovery		Automatic by system		Explicit by user		No recovery



# Record vs. Page References

-  Disc space management
-  Storage management
-  Buffer management

- Within a segment, a *record* could be referenced either directly or through the page it is stored in
- Direct record addressing (spanned record facility)

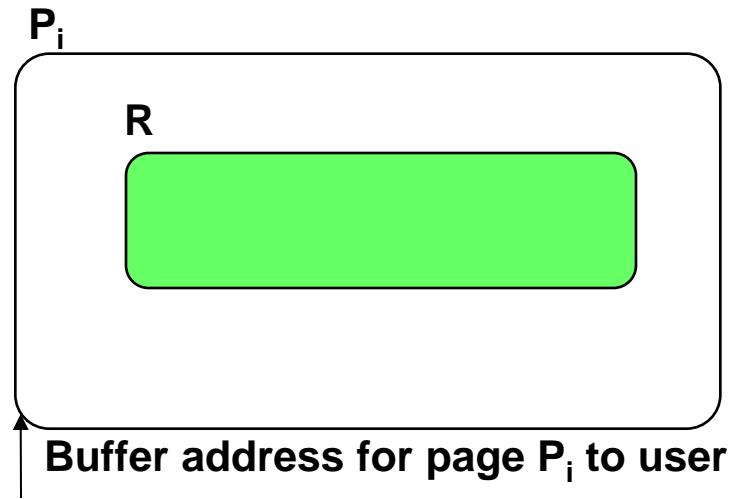


- Complex mapping to pages/blocks, possibly requiring movement of other records (in case of variable length)
- Complex locking and logging



# Page References

-  Disc space management
-  Storage management
-  Buffer management

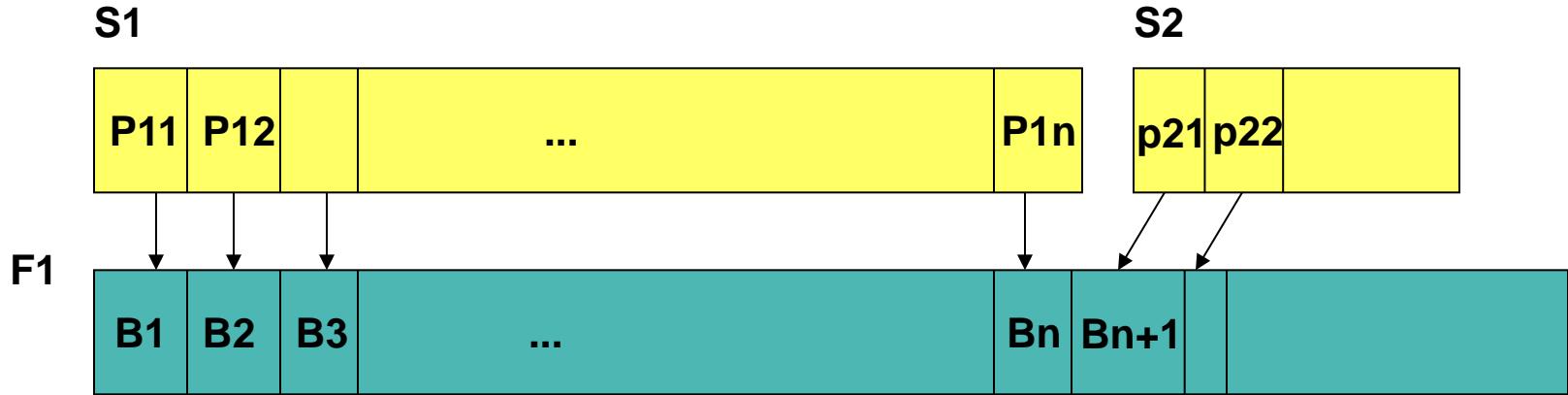


- *Segment* consists of number of pages of fixed size
- Pages have size consistent with block size
- Segment becomes logical linear address space with visible page limits
- Operations are limited to opening/closing of segments and read/write of pages



# Direct Page Addressing

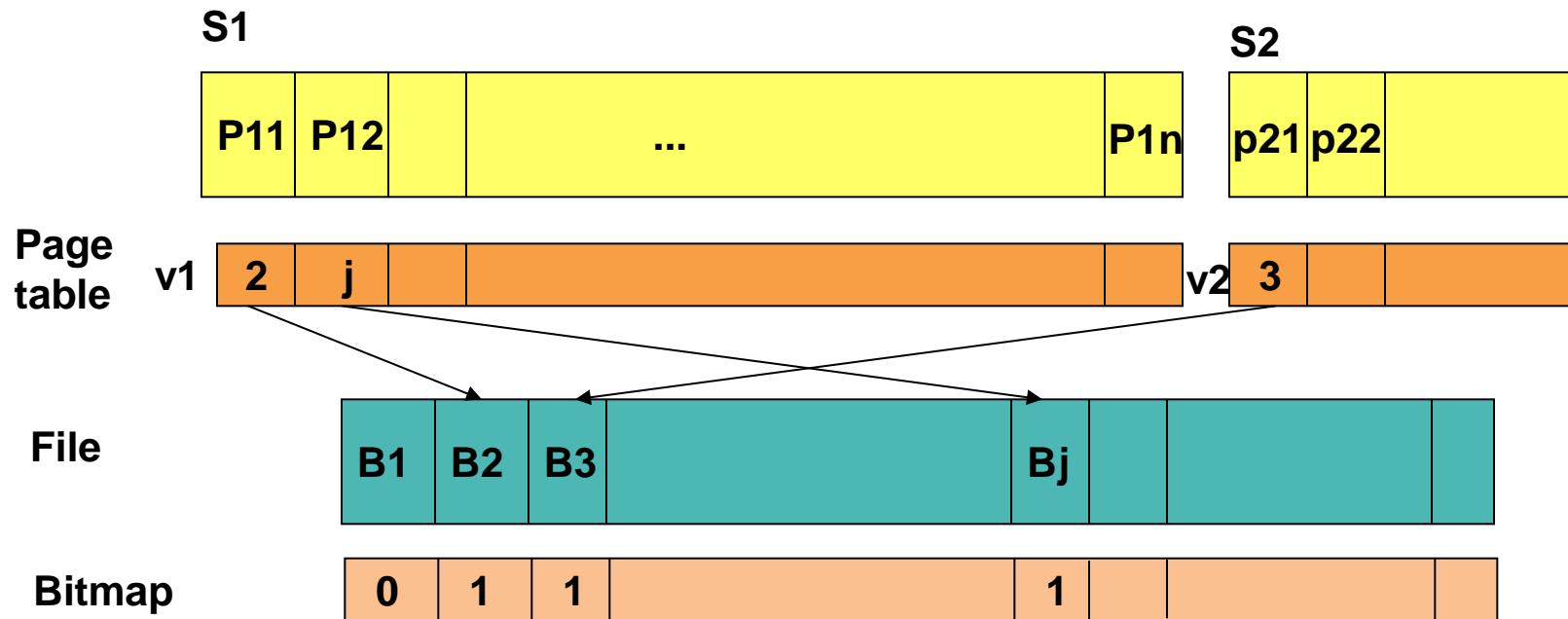
-  Disc space management
-  Storage management
-  Buffer management



- Direct mapping of pages in segments to blocks in files
- Requires reservation of file space at segment-definition time
- Low density when data is slowly inserted



# Indirect Page Addressing



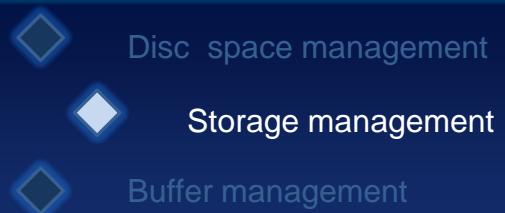
- Better space utilization
- Large page table and bitmaps may have to be mapped to blocks and paged in and out of buffer (may cause additional page faults when requesting a page)

# Update in Place



- The mappings considered so far require that a page be *written back* into the block that was once assigned to it
- Since failures may occur during write-back, this process is considered to be *non-atomic*
- Update in place requires *write-ahead logging (WAL)*
- Before and after images *must be kept* for complete recovery





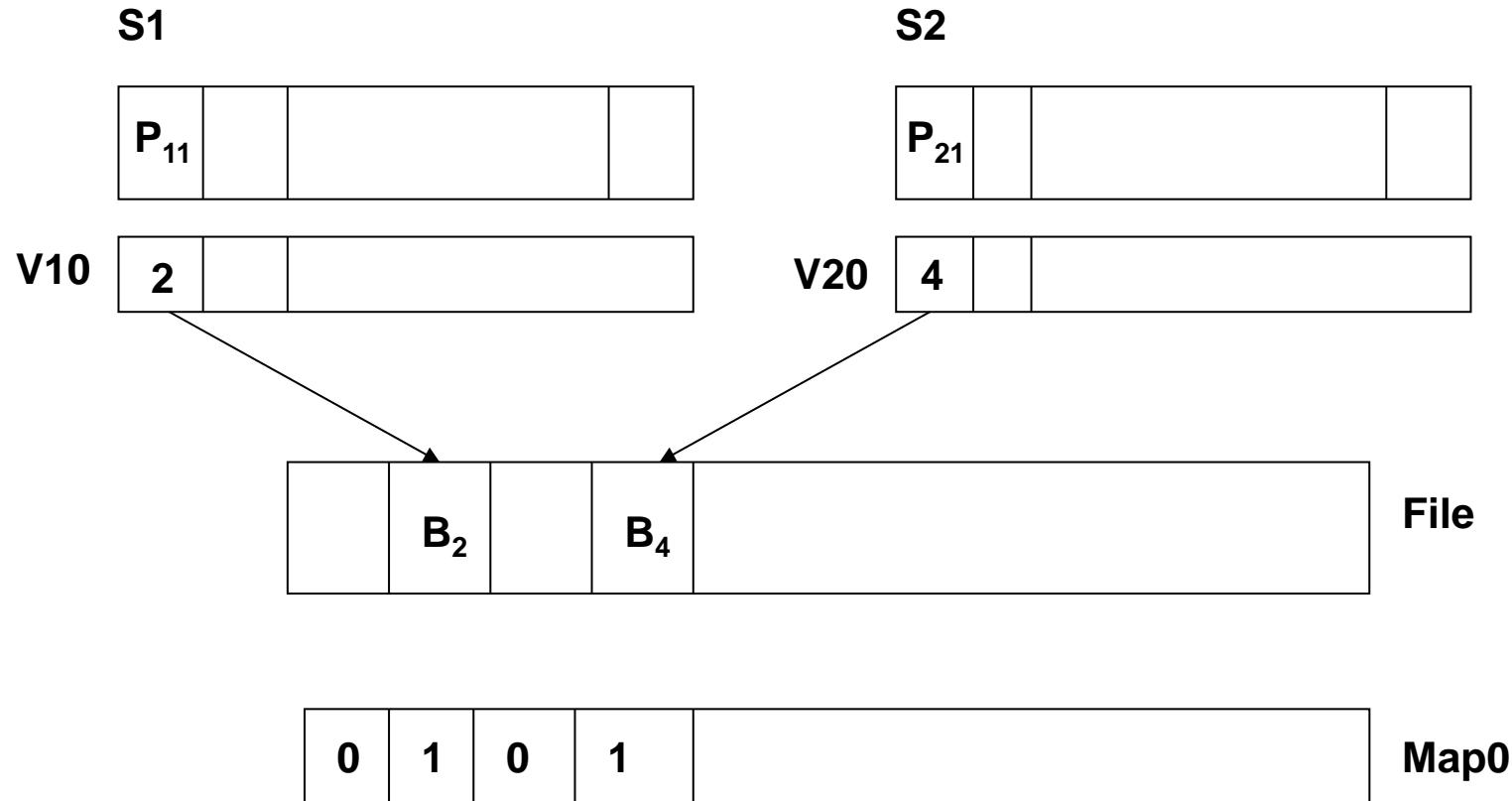
# Shadow pages

- The intention of *shadowing* is to write back a page to another position
- At any given time there are two copies: the old (unchanged) and the new
- The switch from the old to the new copy is considered to be *atomic*

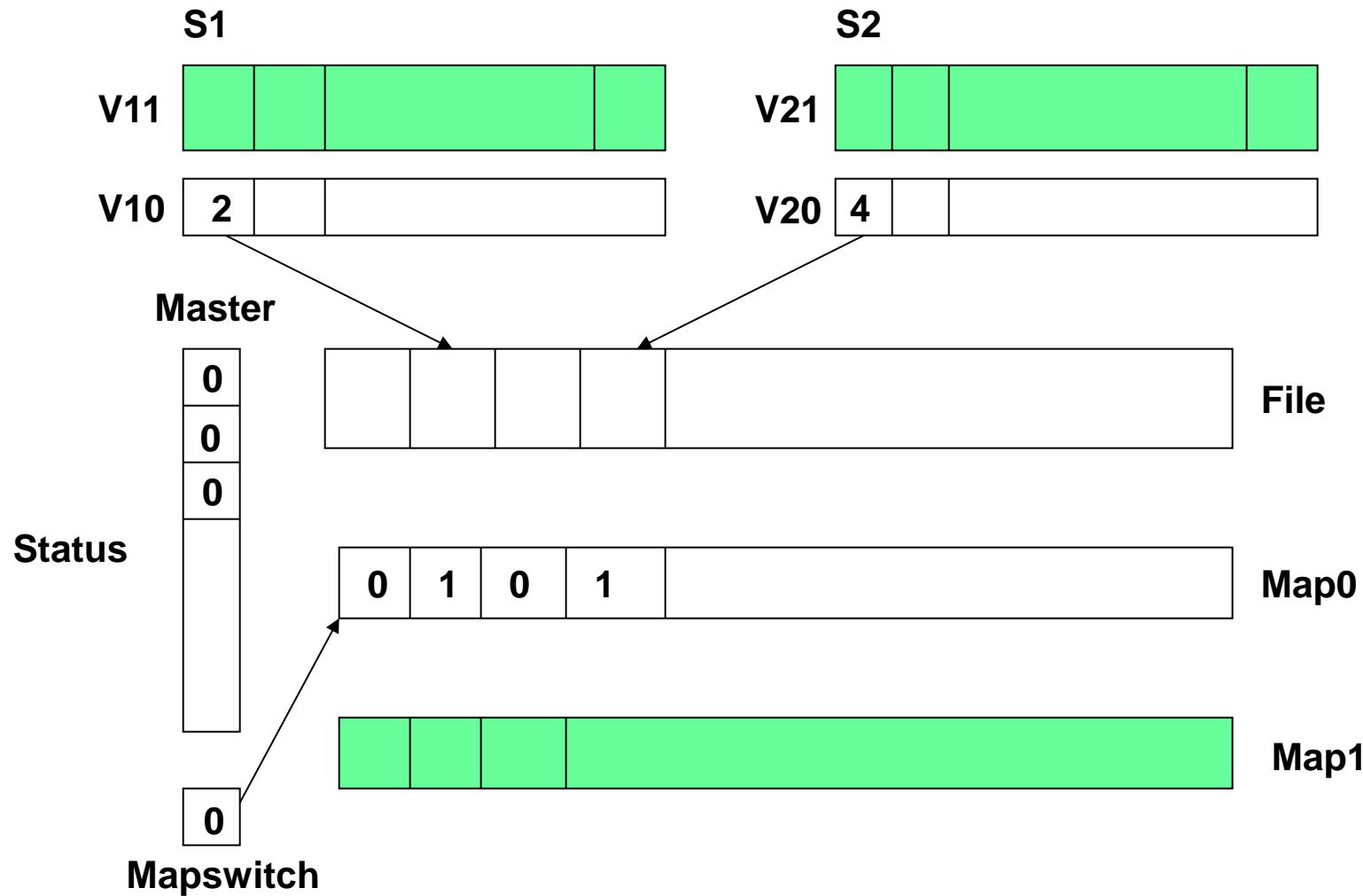


# Shadow Pages (Additional Data Structures)

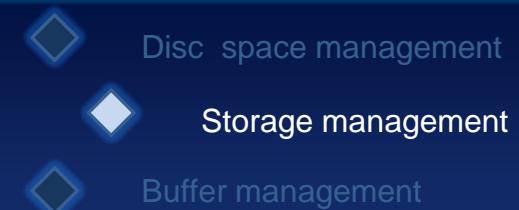
- Disc space management
- Storage management
- Buffer management



# Shadow Pages



# Begin Update with Shadow Pages

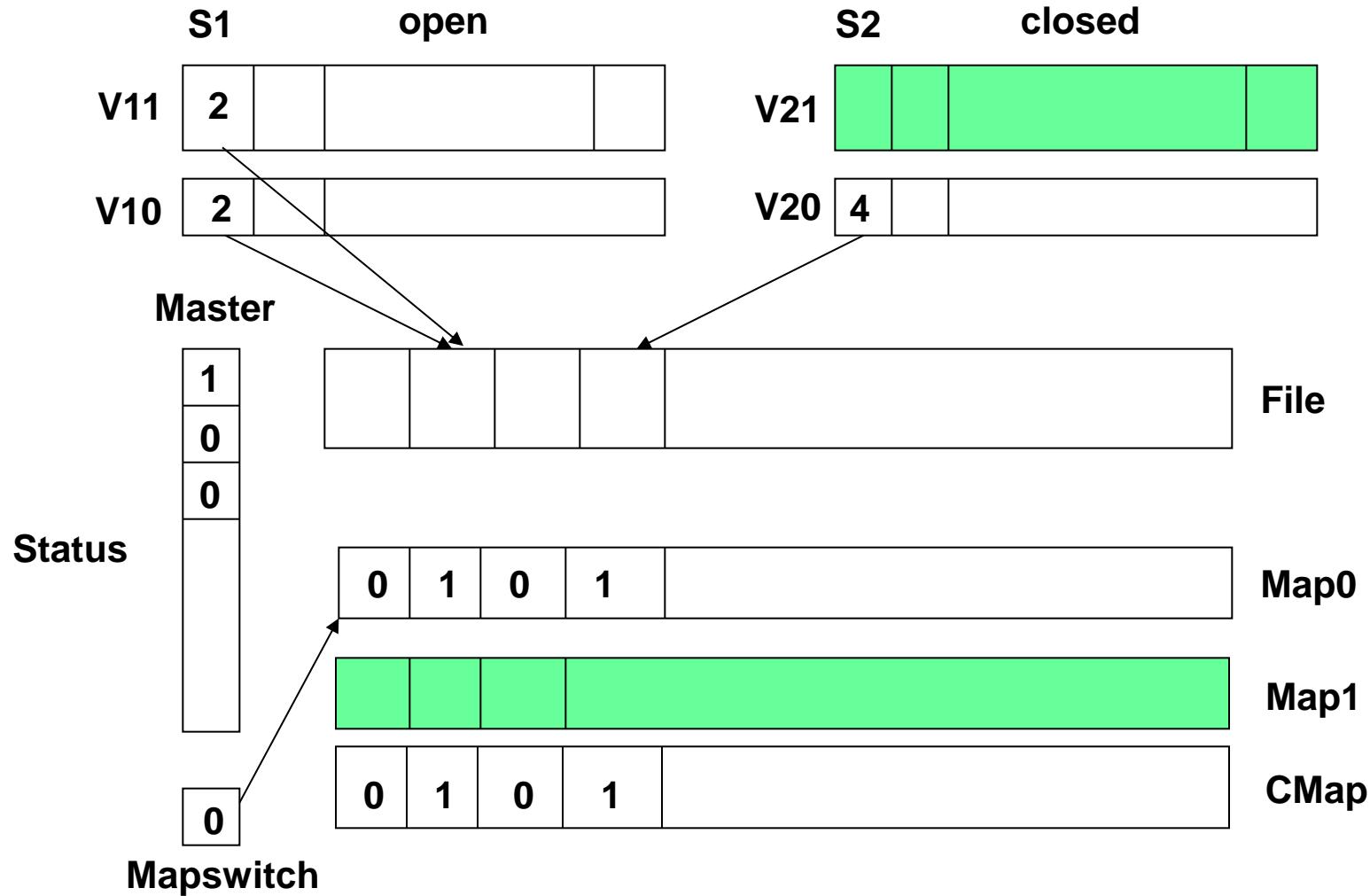


- In the previous diagram dark segments and Map1 (shadows) are not needed since there is no update going on
- *Update interval:*
  - always bracketed by savepoints
  - at begin make shadows of segments that will be updated and of map table
  - update Master and write it atomically to disk
  - make a main-memory copy Cmap of the actual mapping table



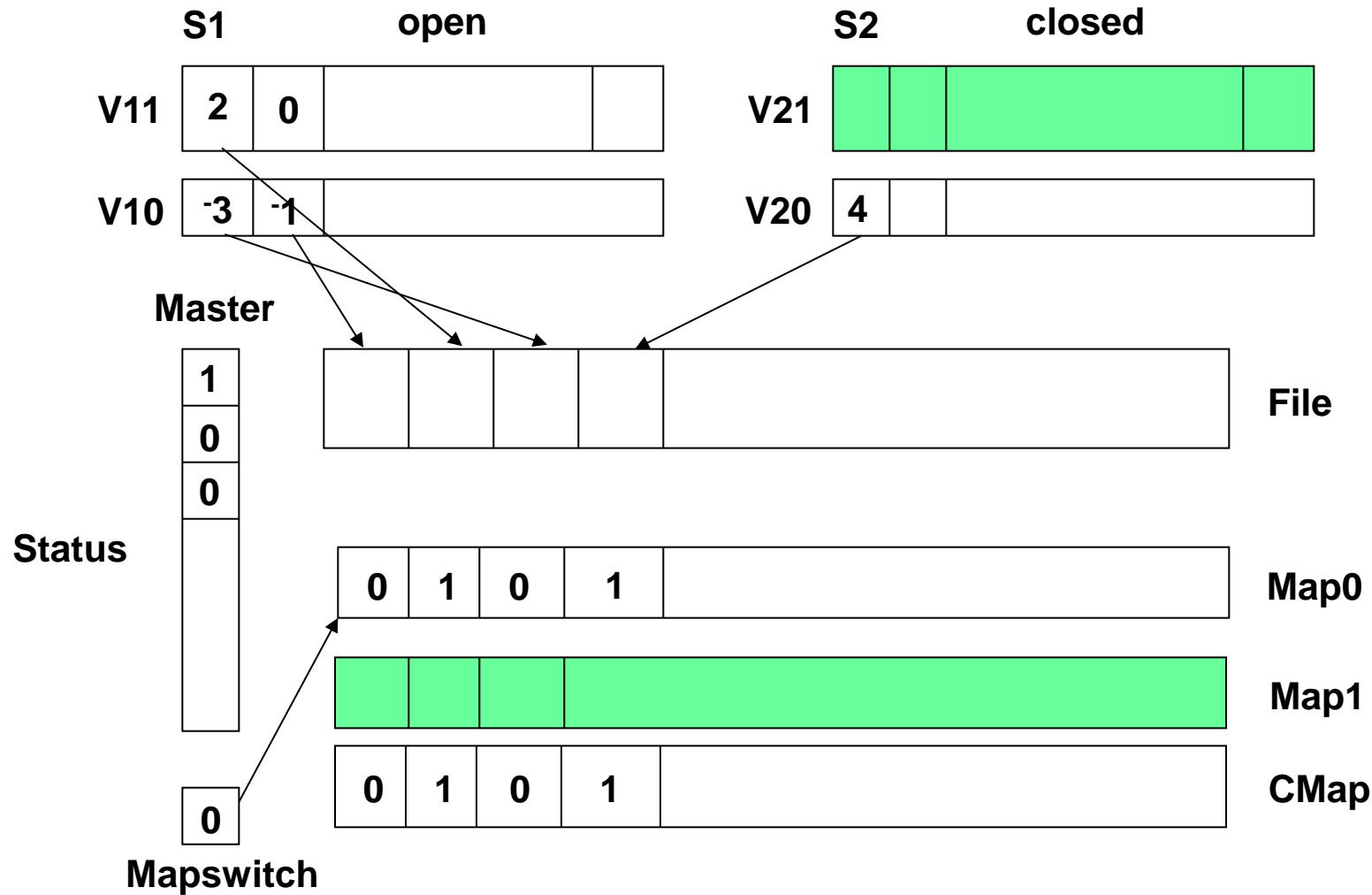
# Begin Update with Shadow Pages

- Disc space management
- Storage management
- Buffer management

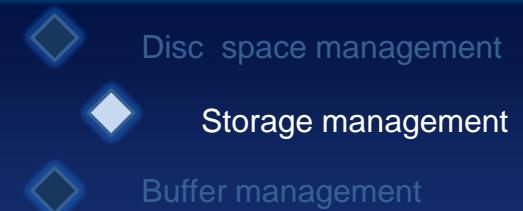


# Update with Shadow Pages (after 2 updates)

- Disc space management
- Storage management
- Buffer management



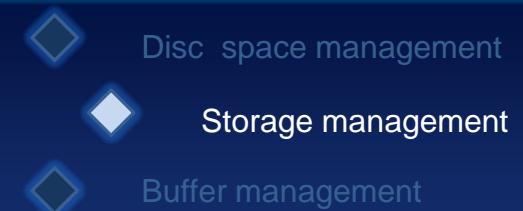
# Finishing Update with Shadow Pages



- Use *Cmap* to generate the new mapping table
- Write new mapping table into unused bitmap (Map1)
- Write  $V_{10}$  (pages of segment  $S1$  that were modified)
- Write all modified pages back to file  $F$
- Change  $Status(1) = 0$  and set  $Mapswitch = 1$  (the new Map)
- Write Master atomically to disk



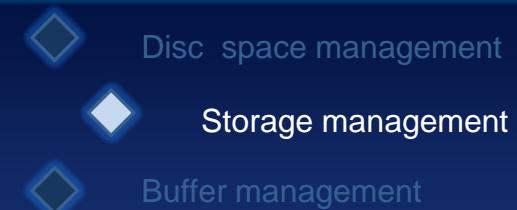
# Checkpointing and Shadow Paging



- *Shadow paging* generates an action-consistent checkpoint if write-back of whole DB is done at once
- Could be restricted to single segment
- *Savepoints/checkpoints* based on segments, not on transactions → recovery is segment based
- To implement transaction-based recovery, logging is needed
- Transaction-consistent checkpoints would be needed to avoid logging (inefficient because of waits)



# Evaluation of Shadow Paging



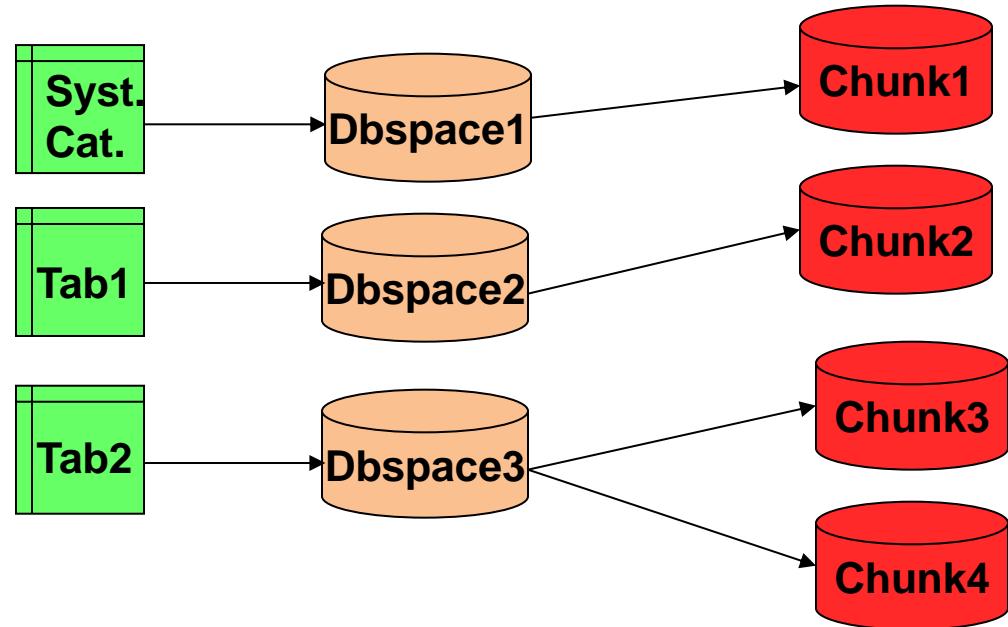
- No WAL needed (w. transaction consistent checkpointing)
  - Rollback to a consistent state is easy (built-in)
  - Since checkpointing generates action-consistent databases, logical logging is feasible
  - Shadow paging destroys clustering (write-back to arbitrary position, could be limited using porosity)
  - Checkpointing causes long delays in on-line ops.
  - Page tables and mapping tables may be large and cause additional page faults
- *update in place* better overall strategy for large DBs



# Logical Storage Units in Informix US



- *Logical units of storage*
  - Dbspaces
  - Blobspace
  - Sbspace
  - Extspace
  - Tblspace
- *Logical units dictated by relational model*
  - Databases
  - Tables
- Dbspaces are the Informix equivalent of segments



# Fragmentation

-  Disc space management
-  Storage management
-  Buffer management

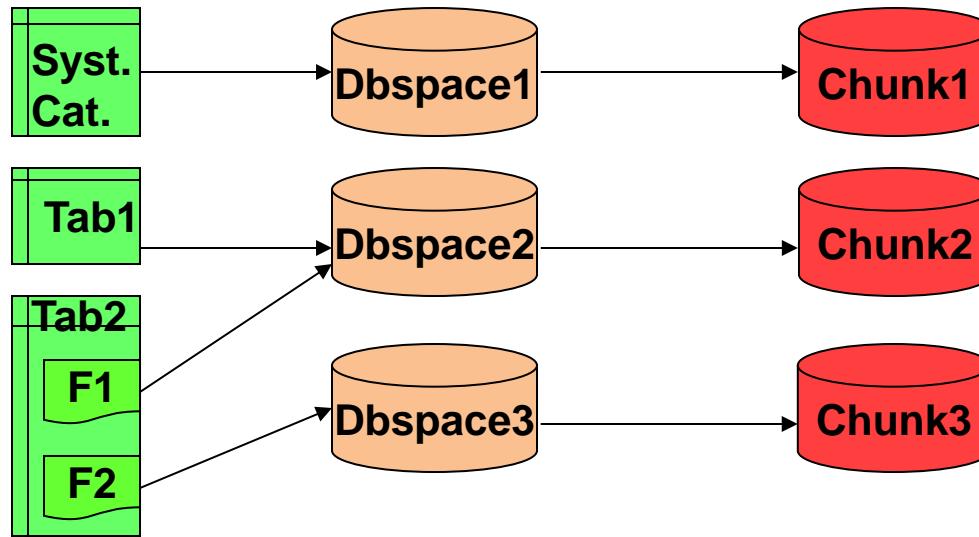
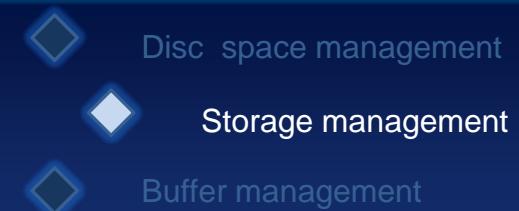


Table fragments can be allocated to different dbspaces:

```
CREATE TABLE tablename ... FRAGMENT BY ROUND  
    ROBIN IN dbspace2, dbspace3;
```

or

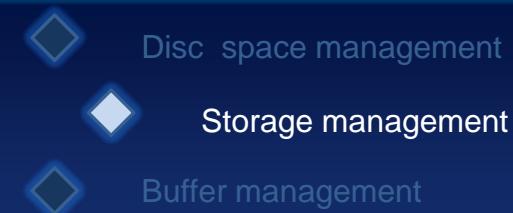
```
CREATE TABLE tablename ... FRAGMENT BY EXPRESSION  
    <Expression 1> in dbspace2,  
    <Expression 2> in dbspace3;
```



# Types of Dbspaces

- **Root Dbspace:**
  - contains all internal tables (e.g. tables tracking other dbspaces)
  - created by system, can be extended later
  - default dbspace for any DB that is not explicitly allocated elsewhere
- **Temporary Dbspace:**
  - may contain only temporary tables
  - is never backed up, always reinitialized upon start-up
  - no logging (neither physical nor logical) → better performance, fewer checkpoints, smaller backups





# Blobspace, Sbspace, Extspace

- **Blobspace:**
  - store simple large objects (BYTE and TEXT)
  - data written directly to disk, never through shared memory
  - blobspace is never physically logged
- **Sbspace:**
  - store only smart large objects (CLOB = character large object) and BLOB (binary large object)
  - smart large objects have read, write and seek properties similar to UNIX standard files
  - smart large objects can be retrieved in pieces, follow isolation modes
- **Extspace:**
  - logical name associated with any device, access method provided by user

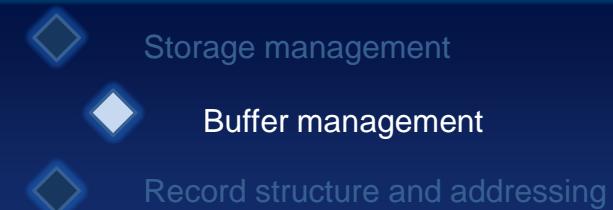


# Buffer Management

- **Tasks of the buffer manager:**
  - provide „infinitely large“ logical, linear address space
  - data is brought into main memory in form of pages
  - data within pages is accessible to instructions of higher layers
  - if requested page is not already in buffer, it must be copied from disk to buffer
  - if no free space is available in buffer, another page must be removed from the buffer
  - if page that is to be removed was changed, it must be written out to disk (with WAL in case of update in place)
- Find page in buffer
- Allocate buffer space to transactions
- Implement replacement strategy



# DBMS vs. OS Buffer Management



- OS *offers* similar functionality (virtual memory mgmt.)
- OS *doesn't offer* quite the right services
  - copying of data between buffers
  - OS uses generic replacement strategies (e.g. LRU)
  - DBMS can do better in selecting replacement strategy (especially for cyclic and tree structured accesses)
  - DB-prefetch is more efficient than OS-prefetch (better determination of reference locality)
  - Selective writing to disk needed in DBMS (e.g. while logging). OS gives no guarantee that write is flushed from OS-buffer to disk.
- DBMS implements its own buffer management in user space

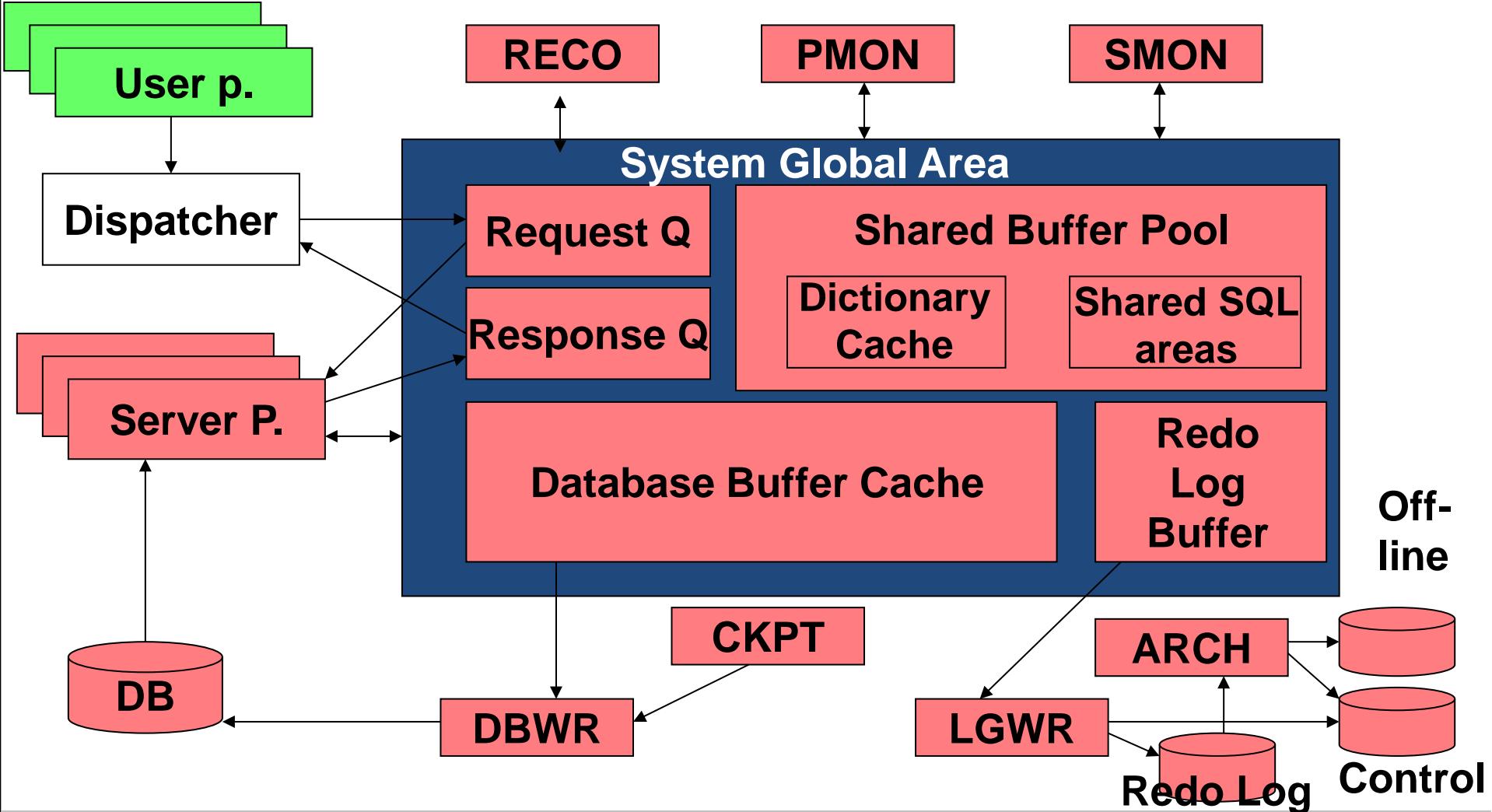


# Oracle Server Architecture

Storage management

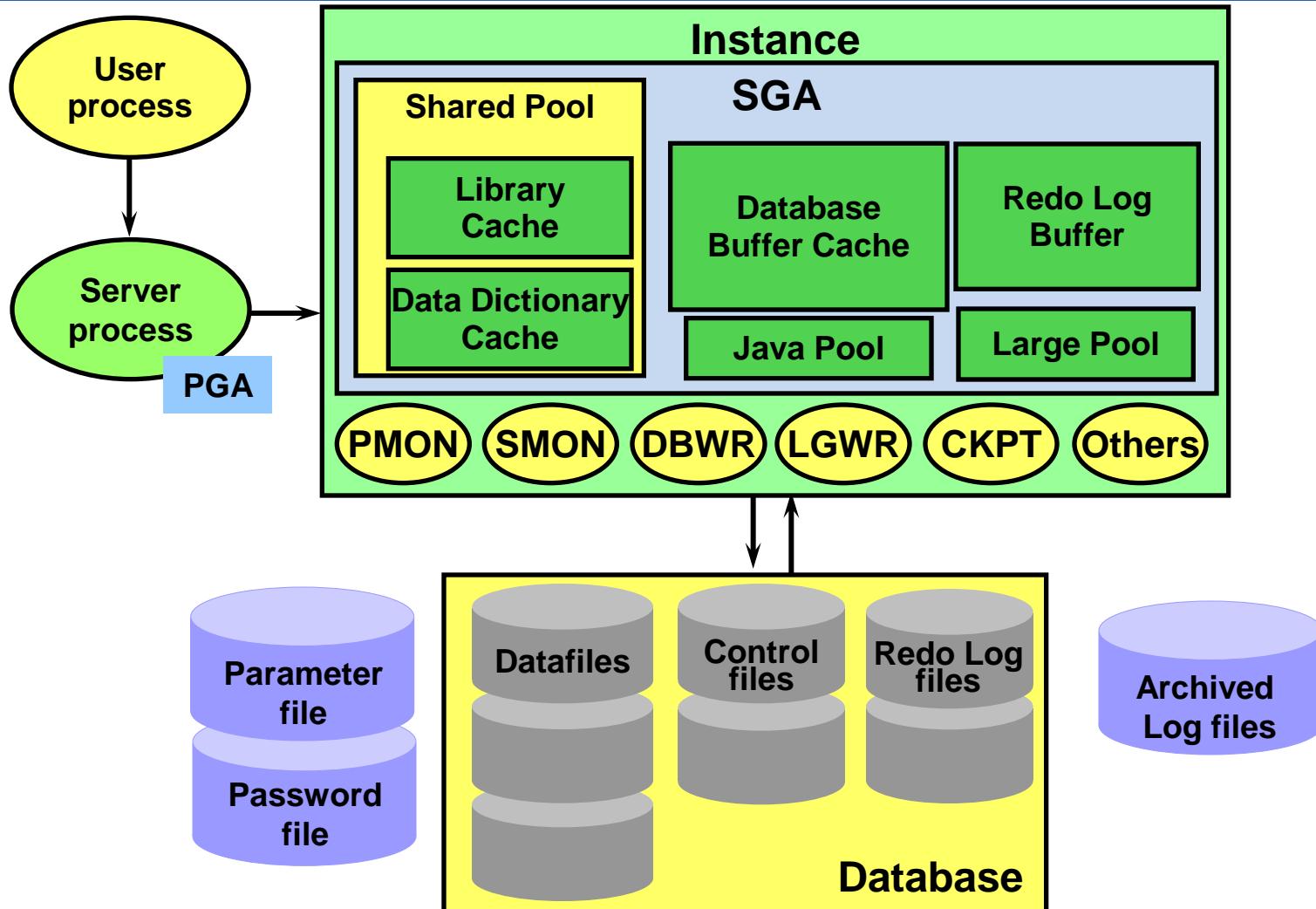
Buffer management

Record structure and addressing



# Oracle Server Architecture

- ◆ Storage management
- ◆ Buffer management
- ◆ Record structure and addressing



# Informix OnLine Shared Memory

Storage management

Buffer management

Record structure and addressing

## Resident portion

SM header	Buffer-hdr T	Chunk table	Mirrored-chunk T	Dbspace T
Lock T	TX-table	User T	Page-cleaner T	Tblspace T
Buffer pool				

## Virtual portion

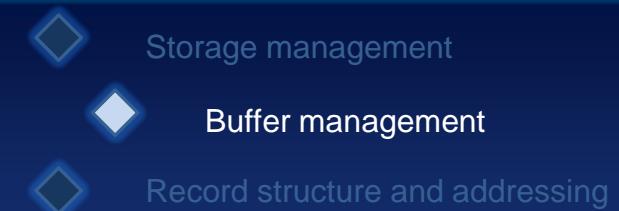
Session structures	Thread structures	Dictionary cache
Thread stacks	Thread heaps	Sorting pool
Big buffers	Global pool	Unallocated memory

## Communications portion

Client/server IPC messages



# Function of DB Buffer: Logical Page Reference



- Higher system layers determine page via system catalogue or access paths
- Pages are requested for reading or writing
  - update requests cause an entry in the buffer control block
- Page is placed in buffer and address is passed to caller
- Page is FIXed in buffer (ensures that page stays in buffer frame for the length of the operation)
- Afterwards page is explicitly freed (UNFIX)
- Page can only be replaced after UNFIX



# Logical Page Reference

- **Page is in buffer (~100 instructions)**
  - locate
  - FIX page (if necessary, write entry into buffer control block)
  - pass address to caller
- **Page not in buffer → physical page reference (~5000 instr.)**
  - buffer search fails
  - find free buffer frame
  - find replaceable page in buffer (if page was modified, write to disk)
  - locate requested page on disk
  - transfer requested page
  - FIX page (if necessary, write entry into buffer control block)
  - pass address to caller



# Page Reference String

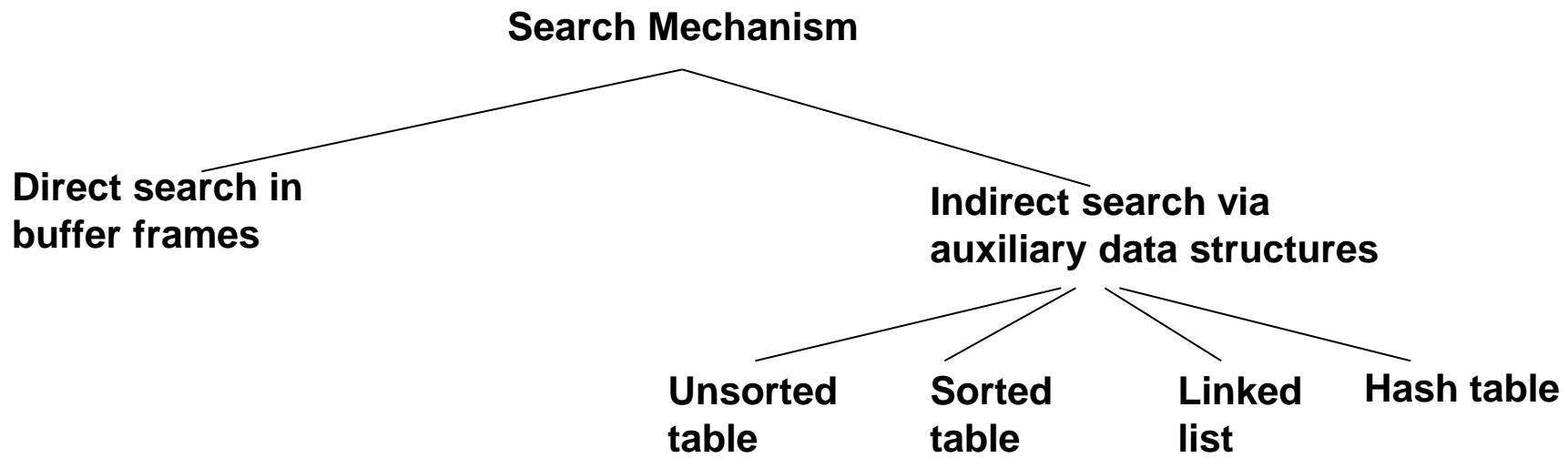
- **Page reference string:** sequence of all logical page references caused by transactions during a certain time interval
  - *transaction mix*
  - *access paths*
- Different logical page reference strings result in different physical page reference strings
- One logical reference string may result in different physical reference strings
  - *replacement strategies*
  - *buffer size*

# Finding a Page

- Storage management
- Buffer management
- Record structure and addressing

**Most frequent operation**

→ must be very efficient

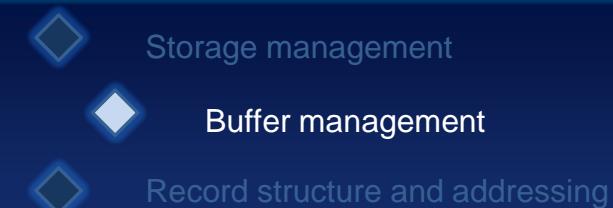


# Finding a Page - direct search

- Each page header in the buffer is checked
  - If *page is in buffer*
    - 1/2 of all pages in buffer checked on average
  - If *page is not in buffer*
    - all pages in buffer must be checked
- In virtual memory systems additional page faults (swapping) could occur
  - not recommended for realistically sized buffers



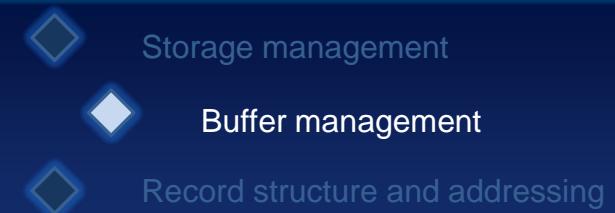
# Finding a page - auxiliary structures



- **Unsorted table**
  - successful  $n/2$  accesses
  - unsuccessful  $n$  accesses
- **Sorted table**  
(maintenance cost)
  - successful  $n/2$  accesses
  - unsuccessful  $n/2$  accesses
  - binary search  $\log n$  accesses
- **Linked list**
  - useful when sequence must be represented
  - low maintenance cost
- **Hash table**
  - low search overhead (worst case = length of overflow chain)
  - typically less than 1.05 accesses on average



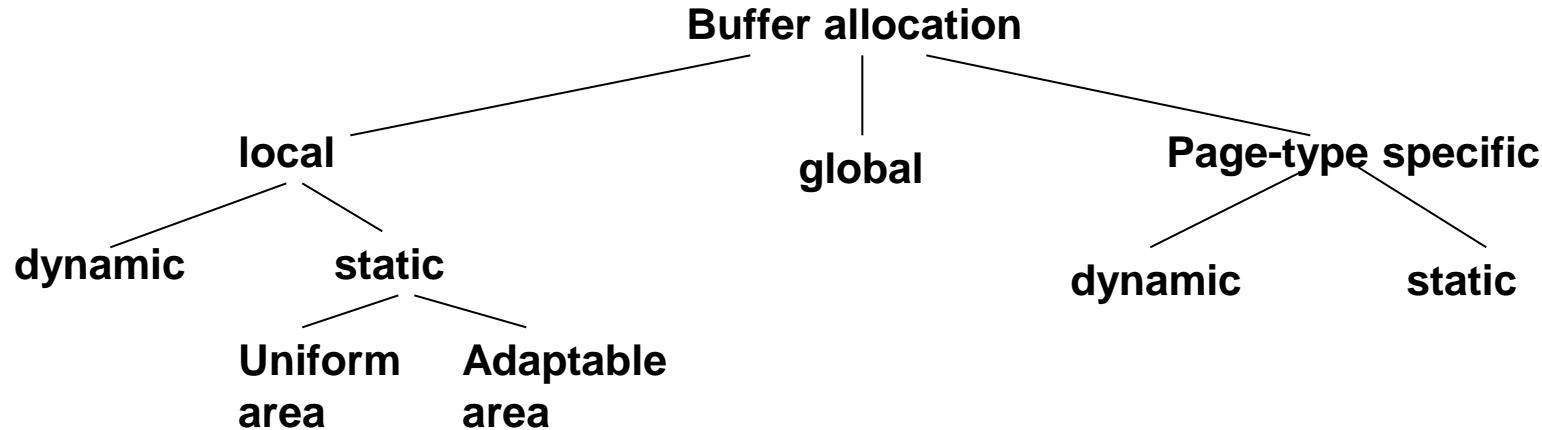
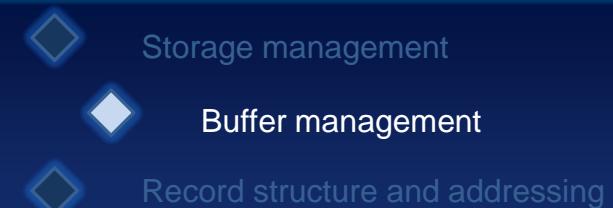
# Buffer Allocation



- Comparable with virtual memory in OS
- **Additional requirements:**
  - several users can access *pages* in the DB-buffer simultaneously
  - locality of reference results from access by multiple transactions to few pages (inter-transaction locality)
  - 3-10% of all pages cause 80% of all accesses
    - catalog, memory and buffer management tables, root pages in tree-structured indexes, log
  - accesses within a single transaction tend to be sequential (intra-transaction sequentiality, inter-transaction locality)
  - access-probability is predictable



# Buffer Allocation Strategies



- **Local strategies:** allocation per transaction
- **Global strategies:** consider the whole transaction workload
- **Page-type specific strategies:** partition the buffer into areas that are exclusively used by access path pages, user data, etc.
- Global strategies are the same as replacement strategies (later)
- Static allocation strategies are impractical with variable transaction load and length
- ➔ only dynamic local strategies discussed now (working set)



# Working Set Allocation Strategy (P. Denning, CACM 11, 5, 68)

Storage management

Buffer management

Record structure and addressing

**Working Set  $W(t, \tau)$**

set of pages that was referenced by a transaction within its last  $\tau$  references measured at time  $t$

$\tau$

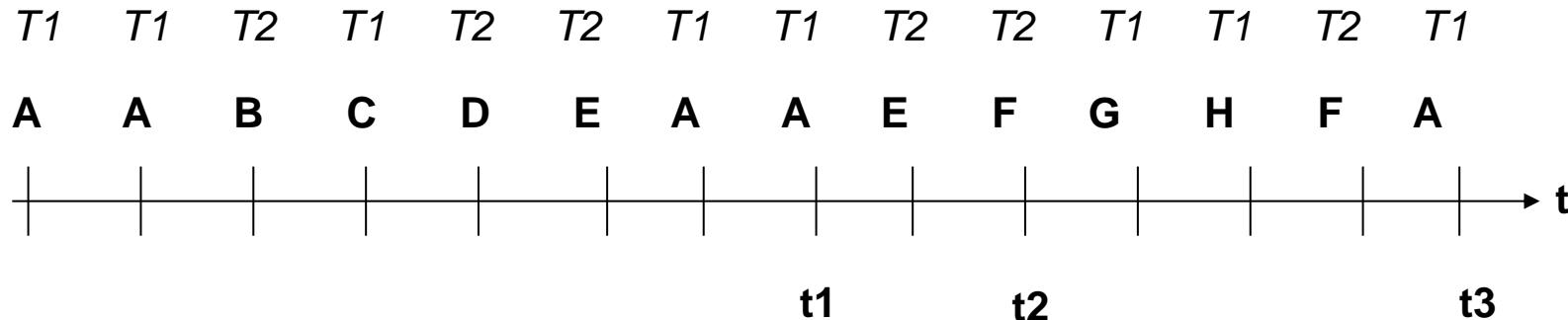
size of window given as number of references

$w(t, \tau) = |W(t, \tau)|$

size of working set at time  $t$



# Example of Working Set Allocation



$W(t, \tau)$  with  $\tau = 5$

- T1:  $W(t_1, 5) = \{A, C\}$
- T2:  $W(t_1, 5) = \{B, D, E\}$
- T1:  $W(t_2, 5) = \{A, C\}$
- T2:  $W(t_2, 5) = \{B, D, E, F\}$
- T1:  $W(t_3, 5) = \{A, G, H\}$
- T2:  $W(t_3, 5) = \{D, E, F\}$

- $w(t_1, 5) = |W(t_1, 5)| = 2$
- $w(t_1, 5) = |W(t_1, 5)| = 3$
- $w(t_2, 5) = |W(t_2, 5)| = 2$
- $w(t_2, 5) = |W(t_2, 5)| = 4$
- $w(t_3, 5) = |W(t_3, 5)| = 3$
- $w(t_3, 5) = |W(t_3, 5)| = 3$

The average size of the WS of a transaction indicates its reference locality

# Working Set Method

- **Principle of operation:**
  - pages in the working set of a transaction should be kept in buffer
  - pages that are no longer in the WS can be replaced
  - when locality is high, WS shrinks
- **Important:**
  - window size  $\tau$  must be specified
  - since working set method doesn't distinguish among the elements in the WS, an additional replacement strategy must be defined
- **Refinement (priorities):**
  - Use  $\tau$  of different sizes based on transaction classes

# Implementation of the Working Set Method



Storage management



Buffer management



Record structure and addressing

For each active Transaction  $T_i$  keep a counter:  $TRC(T_i)$

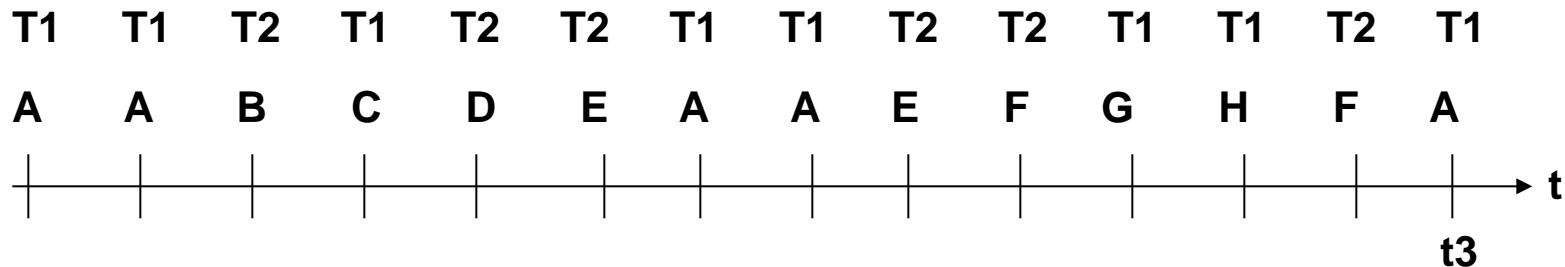
Each page  $k$  in the buffer has a reference counter  $LRC(T_i, k)$  for each transaction that has been active on it

Whenever  $T_i$  references page  $k$ ,

→ Counter  $TRC(T_i)$  is incremented and stored in  $LRC(T_i, k)$

A page may be substituted if for all active transactions:

$$TRC(T_i) - LRC(T_i, k) \geq \tau$$



$$TRC(T_1) = 8 \quad LRC(T_1, A) = 8 \quad LRC(T_1, C) = 3 \quad LRC(T_1, G) = 6 \quad LRC(T_1, H) = 7$$

$$TRC(T_2) = 6 \quad LRC(T_2, B) = 1 \quad LRC(T_2, D) = 2 \quad LRC(T_2, E) = 4 \quad LRC(T_2, F) = 6$$

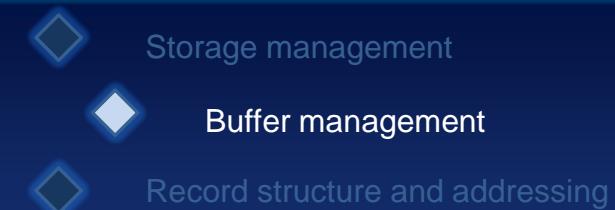
pp. B and C can be replaced



# Replacement Strategies

- Whenever a page fault occurs, one (or more) page(s) must be replaced
- *Problem:* which page to replace in order to obtain globally optimal run-time behavior
- Similar to OS's virtual memory but
  - VM interprets every access individually, i.e. every page is replaceable at any time
  - pages in buffer are FIXed to guarantee writing to same buffer position → multiple accesses to a fixed page are considered as one logical reference

# Prefetching vs. On-demand Fetching



- *Prefetching* gets pages ahead of time → benefit depends on quality of prediction
- *Prefetching* useful for sequential operations (e.g. scan of relation)
- Preplanning useful in “real-time“ environments (pessimistic, upper bounds, generous buffers)
- On-demand fetching gets page when actually needed, may have to displace page already in buffer.

$$TRC(T1) = 8$$

$$LRC(T1,A) = 8$$

$$LRC(T1,C) = 3$$

$$LRC(T1,G) = 6$$

$$LRC(T1,H) = 7$$

$$TRC(T2) = 6$$

$$LRC(T2,B) = 1$$

$$LRC(T2,D) = 2$$

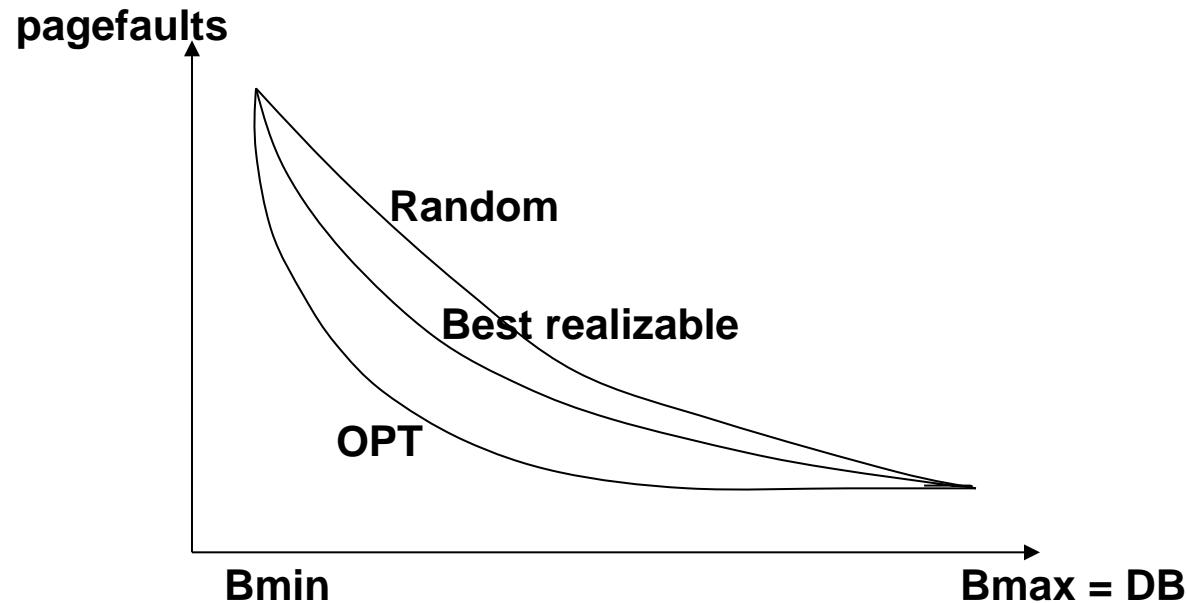
$$LRC(T2,E) = 4$$

$$LRC(T2,F) = 6$$



# Replacement: Upper and Lower Bounds

- Storage management
- Buffer management
- Record structure and addressing



***OPT (Belady's Optimal algorithm):*** best possible strategy determined a posteriori with known reference string

***RANDOM:*** no knowledge at all

***Best realizable:*** between OPT and RANDOM



# Replacement Strategies

- Use past usage patterns to anticipate future use
  - accesses in buffer (all, last, none)
  - age (since loading, since last reference, not at all)
- **FIFO** (First In - First Out)
  - considers only age, replaces the oldest regardless of references
- **LFU** (Least Frequently Used)
  - considers number of references only
  - reference counter incremented by each access
  - bad because old but once frequently used pages can't be replaced

# LRU Replacement Strategies

- **LRU (Least Recently Used)**
  - considers age and references
  - replaces page that hasn't been referenced the longest
  - interpretation of the word “used”
    - Least Recently Referenced
    - Least Recently Unfixed
  - since logical reference may contain many (sometimes  $O(10^3)$ ) physical accesses, the interpretation as Unfixed is more accurate
- **LRU - k**
  - averages over the last k references (reference density)
  - additional work in determining reference density
  - LRU-2 offers best cost/performance ratio



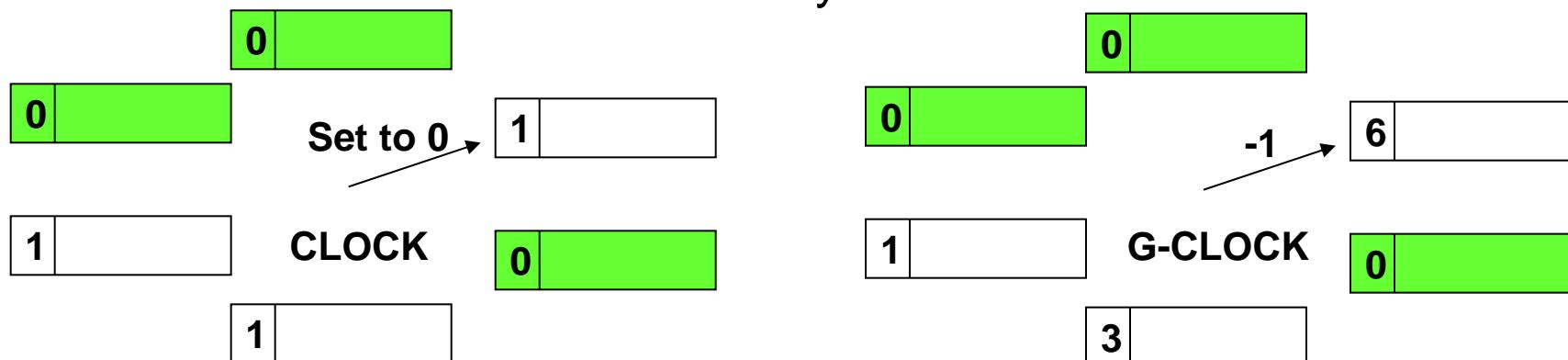
# CLOCK Replacement Strategies

## ▪ CLOCK

- each page has a reference bit that is set when page is accessed
- when replacement is needed, pointer goes around and first page with reference bit = 0 is selected
- after passing by a page, reference counter is set to 0

## ▪ G-CLOCK

- like CLOCK but with reference counter incremented/decremented by 1



# Recent Developments

- How can we combine the advantages of different algorithms without paying the penalties?
- Adaptive Replacement Cache (ARC)
  - Patented by IBM 2006
  - Description taken from Wikipedia  
[http://en.wikipedia.org/wiki/Adaptive\\_replacement\\_cache](http://en.wikipedia.org/wiki/Adaptive_replacement_cache)



# Adaptive Replacement Cache - ARC

- LRU maintains an ordered list of pages based on the time of most recent access.
- New entries are added at the top of the list, after the bottom entry has been evicted. Cache hits move to the top, pushing all other entries down.
- ARC improves the basic LRU strategy by splitting the cache directory into two lists, T1 and T2, for recently and frequently referenced entries.
- T1 and T2 are extended with a *ghost list* ( $B_1$  or  $B_2$ ),  
*which is attached to the bottom of the two lists.*
  - *Ghost lists contain metadata only*
  - *Ghost lists keep track of data usage even after eviction*



# ARC (cont.)

- *The combined cache directory is organized in four LRU lists:*
  - *T1, for recent cache entries.*
  - *T2, for frequent entries, referenced at least twice.*
  - *B1, ghost entries recently evicted from T1 cache, still tracked.*
  - *B2, similar ghost entries, but evicted from T2.*
  - *T1 and B1 together are referred to as L1, a history of recent single references.*
  - *L2 is the combination of T2 and B2.*



# ARC Visualization

```
... [ B1 <-[ T1 <-!-> T2 ]-> B2 ] ...
[..... [.....!..^.....].....]
[ fixed cache size (c) ]
```

The inner [ ] brackets indicate actual cache, which although fixed in size, can move freely across the B1 and B2 history.

L1 is now displayed from right to left, starting at the top, indicated by the ! marker.

<sup>^</sup> indicates the target size for T1, and may be equal to, smaller than, or larger than the actual size (as indicated by !).

New entries enter T1, to the left of !, and are gradually pushed to the left, eventually being evicted from T1 into B1, and finally dropped out altogether.

Any entry in L1 that gets referenced once more, gets another chance, and enters L2, just to the right of the central ! marker. From there, it is again pushed outward, from T2 into B2.

Entries in L2 that get another hit can repeat this indefinitely, until they finally drop out on the far right of B2.



# ARC Replacement

## Replacement

- Entries (re-)entering the caches T1 and T2 will cause ! To move towards the target marker ^
- If no free space exists in the cache, this marker determines whether either T1 or T2 will evict an entry.
- Hits in B1 increase the size of T1 pushing marker ^ to the right, the last entry in T2 is evicted into B2
- Hits in B2 will shrink T1 pushing the marker ^ back to the left and the last entry in T1 is evicted into B1



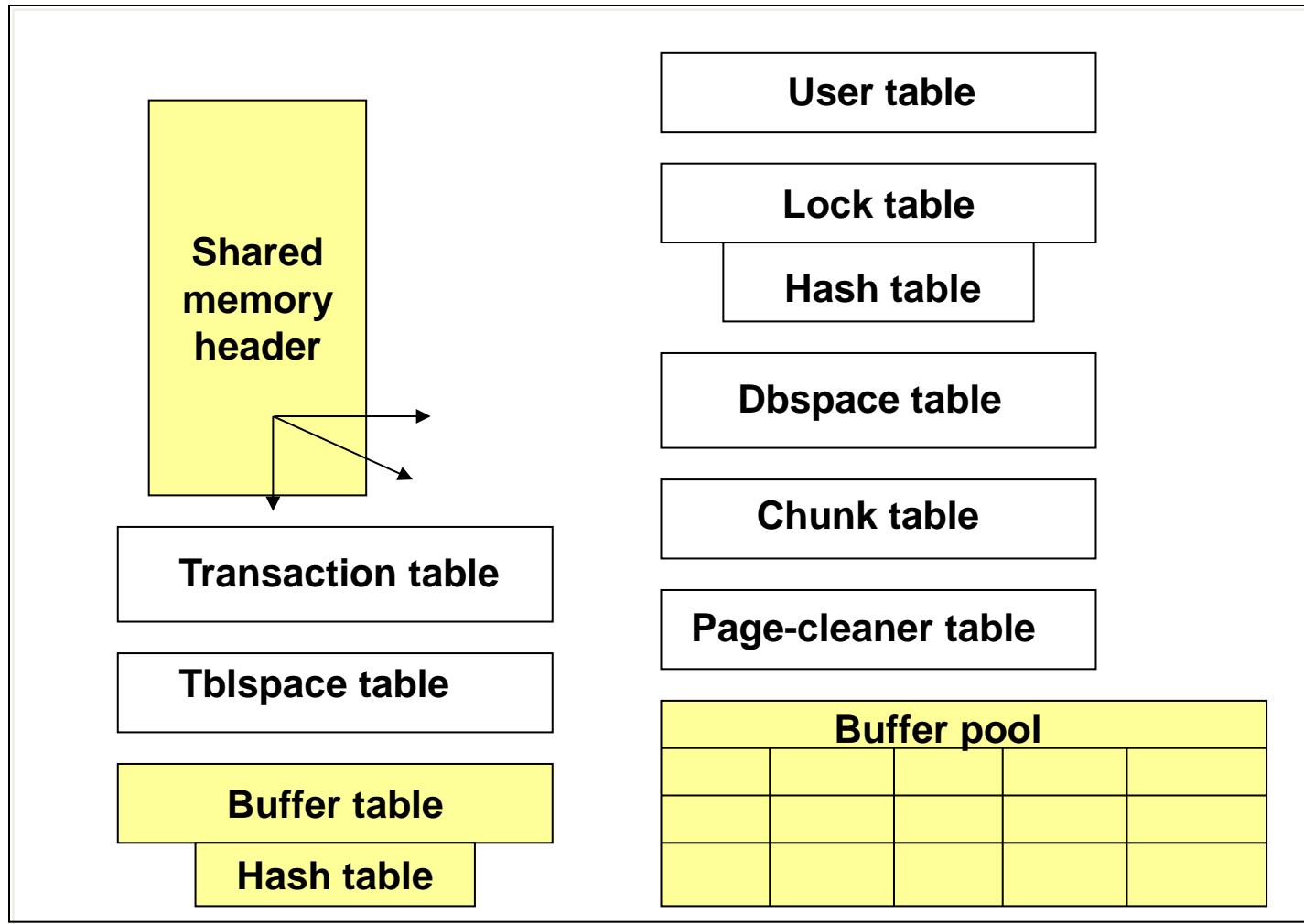
# Informix Buffer Management

## Resident Portion of Shared Memory

Storage management

Buffer management

Record structure and addressing

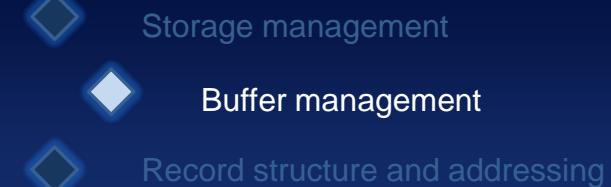


# Buffers

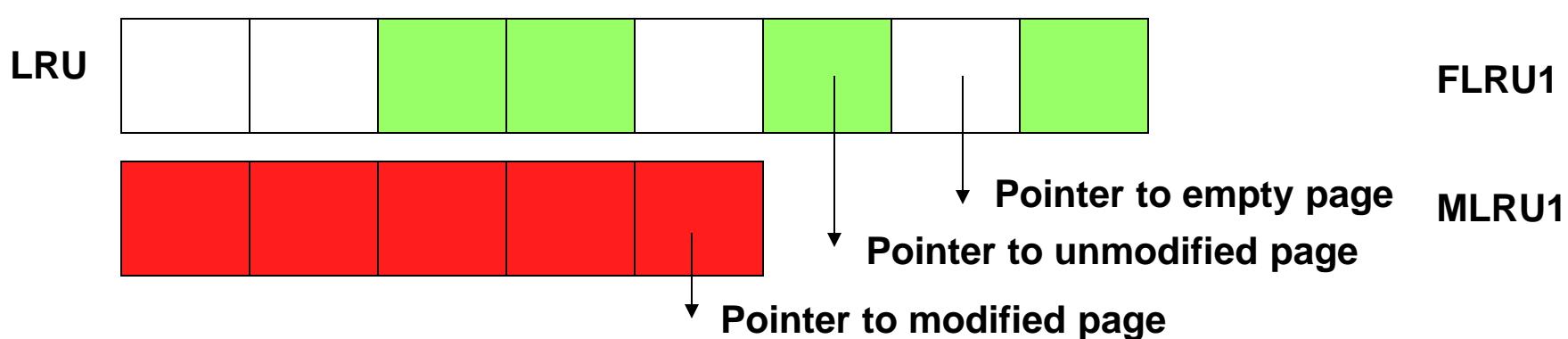
- **Buffer pool** in resident portion of shared memory contains the regular buffers that store database pages
- *Regular buffers* store dbspace pages read from disk
- **Default:** 1000 buffers (buffer is meant as buffer frame containing a page), at least 4/user thread
- Each regular buffer is the size of one DB server page
- I/O is performed in full-page units
- Status of regular buffers is tracked through *buffer table*
  - buffer status (empty, unmodified, modifiedlevel)
  - current lock-access (shared, exclusive)
  - threads waiting for that buffer
- Regular buffers are organized into LRU buffer queues



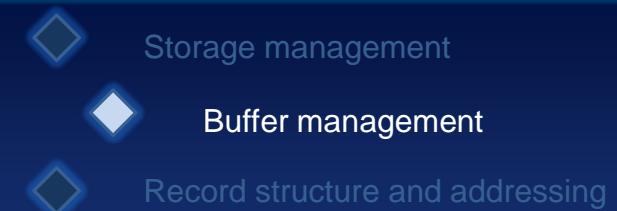
# LRU Queues



- Each buffer tracked through several linked lists of pointers to the buffer table (LRU queues)
- *LRU queue* composed of two queue types:
  - one list tracks free or *unmodified pages* (FLRU)
  - one list tracks *modified pages* in queue (MLRU)
  - separate lists eliminate need to search a queue for a free or unmodified page



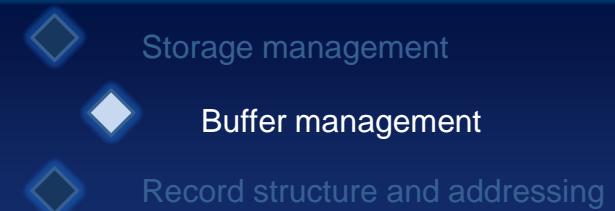
# LRU Queues and Buffer Management



- Pages maintained in **Least Recently Used** order in Qs
- Each buffer (frame) has one entry in one of the buffer Qs
- Buffers are evenly distributed across FLRU queues
- Universal Server randomly selects one FLRU queue and tries to latch the oldest (LRU) page
- If the FLRU queue is locked and the end page cannot be accessed, another FLRU queue is selected
- A specific page is identified by LRU queue location through the control information stored in buffer table
- After a page was used and released, it is placed at the MRU end of the FLRU queue (unmodified) or MLRU queue (mod.)



# Cleaning LRU Queues



- LRU queues are cleaned by Cleaner Threads
- When configuring an LRU queue, parameters **LRU\_MIN\_DIRTY** and **LRU\_MAX\_DIRTY** are specified
- These parameters control the frequency with which data is flushed to disk
- LRU writes are performed as background writes by cleaner threads when percentage of dirty buffers exceeds percentage specified in **LRU\_MAX\_DIRTY**
- LRU writes may also be triggered by foreground writes who alert the cleaner thread of the write
- Cleaning LRU queues in background avoids having to wait for a clean page when a transaction needs it



# Locating a page

- *Thread* first attempts to locate requested page in shared memory
  - acquires mutex on hash table
  - if entry for page is found, try to acquire mutex on buffer entry in buffer table
  - test lock-access level for buffer, if compatible gain access, otherwise requesting thread puts itself on wait queue for buffer
- If page must be *read from disk*
  - locate usable buffer in FLRU queues
  - obtain mutex on queue and use buffer at LRU-end of queue
  - if another thread has mutex, try another FLRU queue
  - remove buffer from FLRU queue and create entry in buffer table

# Releasing a Buffer

- When thread is *finished using* the buffer
  - release the buffer lock
  - wake up first thread waiting for this buffer (if any)
- Releasing an *unmodified buffer*
  - get mutex on buffer table,
  - look for sleeping threads waiting for buffer (priority-ordered)
  - if no thread is waiting, return page to its original FLRU queue if possible, else to any other queue at the MRU-end

# Releasing a Modified Buffer

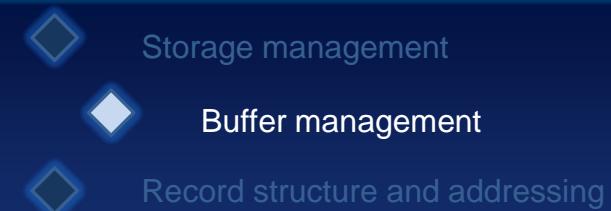
## ■ Releasing a modified buffer

- acquire mutex for buffer, change lock-access type to exclusive
- make sure a before-image exists (e.g. checkpoint) or write it to physical log buffer
- make changes to buffer
- record transaction record in logical log buffer
- release mutex for buffer
- releasing thread acquires mutex on buffer table
- update time stamp in buffer header to match timestamp on buffer
- release lock and append buffer to MRU-end of LRU queue

# Flushing Data to Disk

- When a thread modifies data in a buffer, it marks it as *dirty*
- After buffer is *flushed* to disk, it is marked as not dirty and can be overwritten
- Buffers are flushed by the *Cleaner Threads*
- *Flushing* of regular buffers, physical log buffers and logical log buffers must be synchronized
- Physical log buffers containing before images must be flushed to the physical log on disk before the corresponding regular buffers can be flushed

# Disk I/O Virtual Processors



- **Virtual processors** that perform disk I/O
  - CPU -- where available KAIO (kernel asynchronous I/O)
  - AIO (asynchronous I/O)
  - PIO (physical-log I/O)
  - LIO (logical-log I/O)
- **Reminder:** virtual processor = multithreaded US process that runs multiple threads, one thread at a time until it yields
- *KAIO* performed by CPU virtual processors, for everything that resides on raw disk space
- *AIO* performs non-logging I/O to cooked disk space
- *PIO* and *LIO* perform logging I/O to cooked disk space or if KAIO is not available



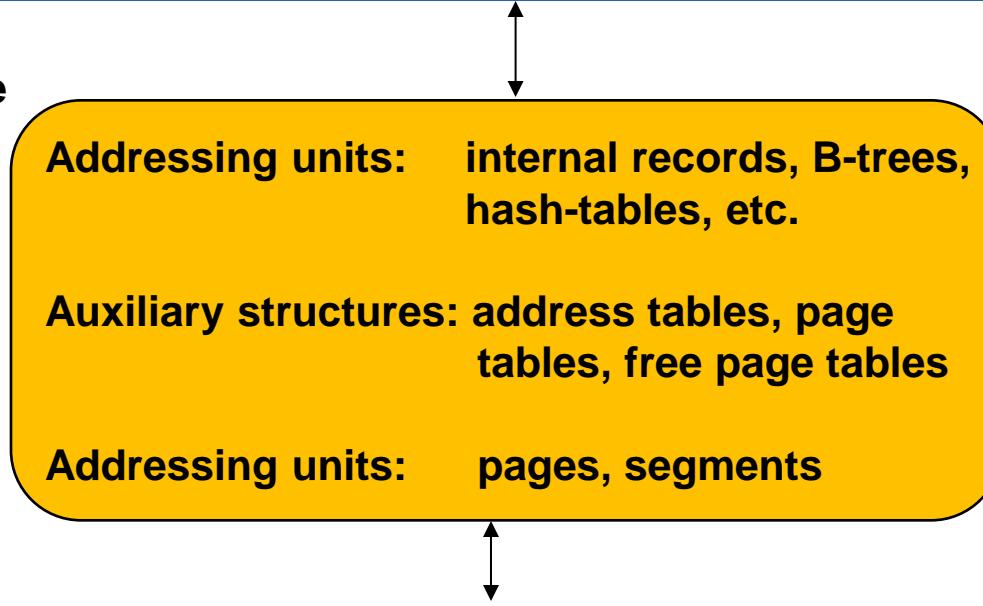
# I/O Priorities

- I/O is executed according to these *priorities* (independent of whether KAIO in the CPU VP is used or other I/O VPs)
  - 1st** logical-log I/O needed for recovery
  - 2nd** physical-log I/O before images since last checkpt.
  - 3rd** database I/O
  - 3rd** page-cleaning I/O
  - 3rd** read-ahead I/O
- KAIO implemented by running a KAIO thread on CPU VP. KAIO thread executes I/O by making a system call to OS, better performance than AIO because no VP-switch.
- AIO if no KAIO available. Each named chunk has own queue, I/O requests ordered to minimize disk head movement

# Storage Structures and Logical Access Paths

- Buffer management
- Record structure and addressing
- Access paths and indexing

Internal interface  
store record, update  
index, etc.

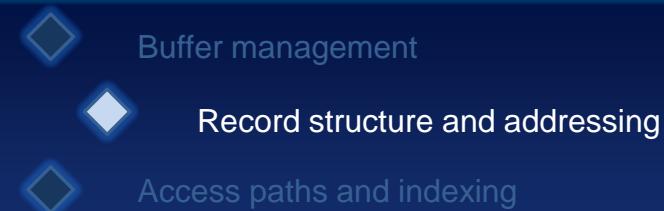


DB-buffer interface

- **Two main components** at this level:
  - *Record manager* responsible for mapping to physical records
  - *Access path manager* responsible for implementation and management of index structures



# Record Manager



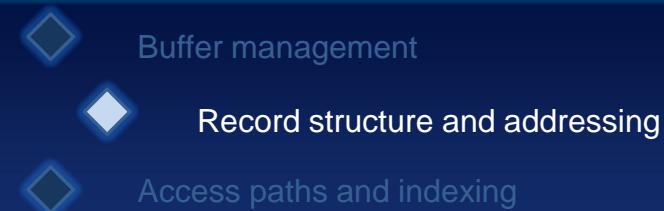
- On insert RM identifies page with enough free space
- Each segment has a **Free-Space-Table  $F$**
- $F$  consists of entries  $f_i$  with length  $L_f$  for each page  $P_i$
- $F$  may require several pages, located at predictable location (usually beginning or end of segment)
- $k$  entries  $f_i$  with length  $L_f$  can be stored in a page where  
$$k = \lfloor (L_p - L_{ph}) / L_f \rfloor \quad \text{and } L_p = \text{page length}$$
$$L_{ph} = \text{page header length}$$

$L_f = 2$  Bytes w. precise free space management

$L_f = 1$  Byte w. fuzzy free space mgmt., multiples of  $\lceil L_p / 256 \rceil$

- Page header contains size and location of available space

# Addressing a Record on Disk



- Each record on disk must be *addressable*
- Each record is located in a page of a segment which is mapped to a block on disk
- The record identifier must allow for the location of a record and for possible movement of pages and segments (indirection)
- An address consists of a **TID** (*tuple identifier*), a **SID** (*segment identifier*) and a **RID** (*relation identifier*)
- SID and RID are often inferred from catalogue information (segment table, chunk table)



# Tuple Identifiers

**Goal:** to allow fast location of a record/tuple and movement within a page.

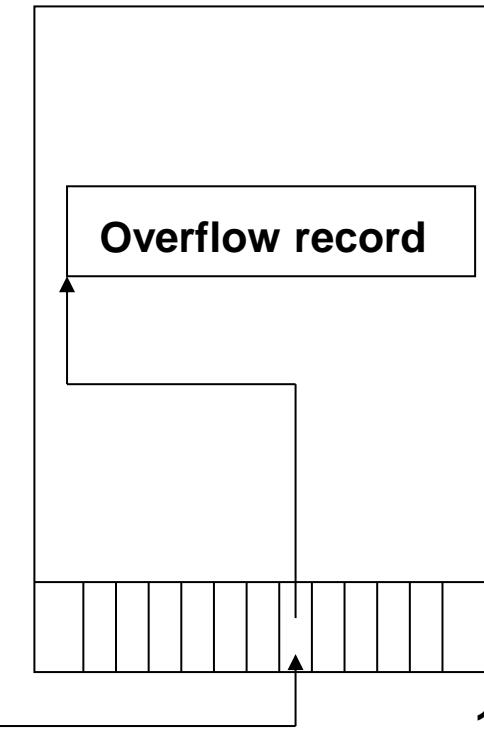
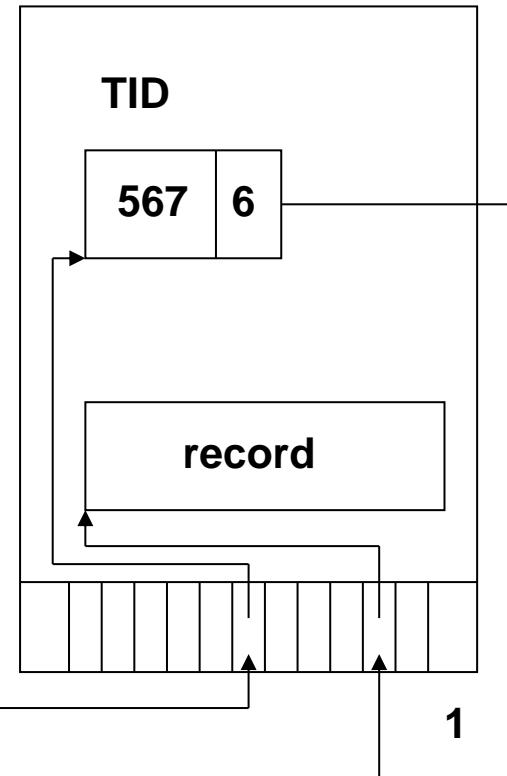
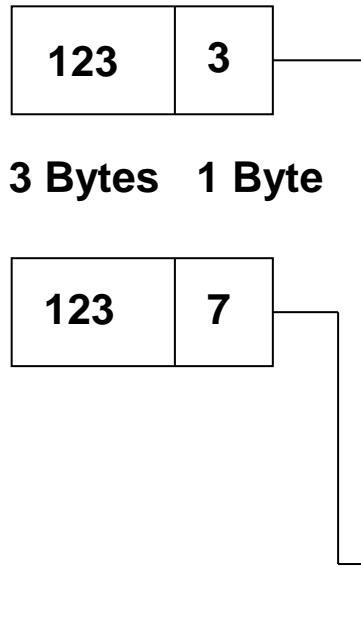
- TID consists of page number (within a segment) and page index position.
- Entry in page index indicates the position of the tuple within the page
  - allows movement within a page (due to own growth/shrinking or that of other tuples on the page) without changing the TID
  - if tuple grows to the point where it can't fit on its home page, a tombstone (TID that points to new location) is set
  - at most one tombstone is set for each tuple: 2nd displacement returns tuple either to home page or changes original tombstone to new TID → each record is located with at most 2 accesses



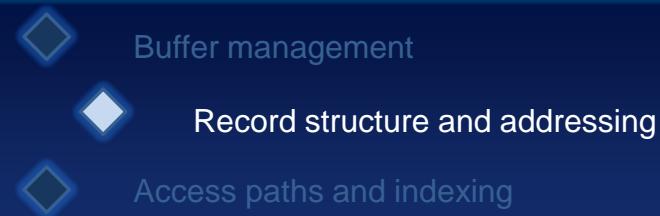
# TID Schematic

- Buffer management
- Record structure and addressing
- Access paths and indexing

## TIDs



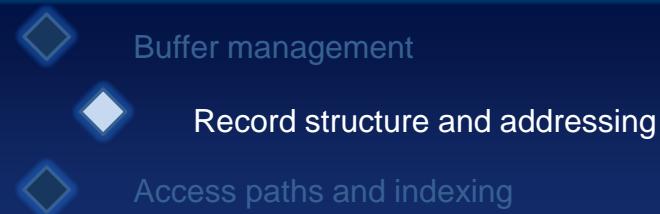
# Object Identification



- **OID:** unique object identifier, immutable for life of object
- two basic implementation possibilities:
  - *surrogates*
  - *structured addresses*



# Object Identification



- **Surrogates:**
  - address is determined via OID in hash-table
  - hash-table usually quite large
  - objects are easily movable (e.g. when they grow)
- **Structured addresses**
  - typical format: DB# | Seg# | Offset
  - mixture of physical and logical addressing
  - difficult to move (off-line, tombstones)
  - efficient access, usually through single disk access
- *Trade-off:* retrieval performance vs. flexibility
- 64-bit OIDs necessary but negative perform.

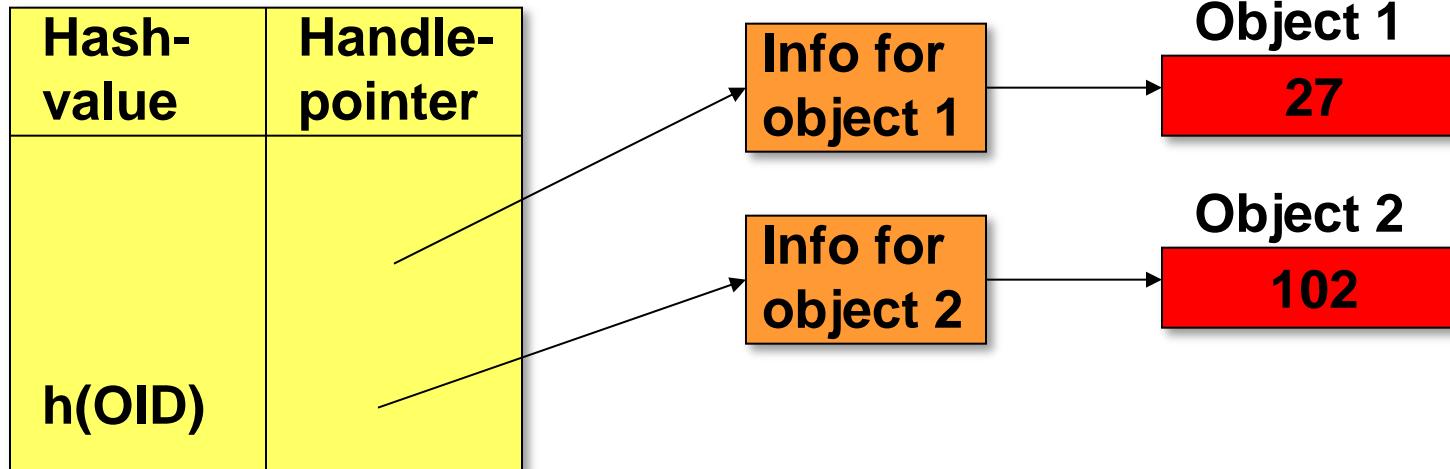


# Referencing Objects in the Application's Address Space

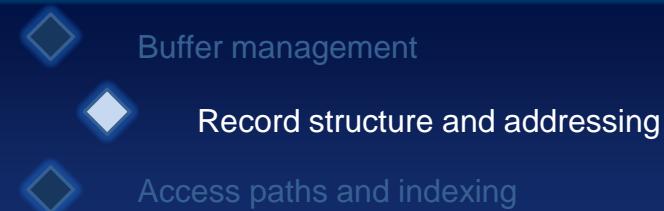


- **Main problem:** fast following of references
- Two basically *different options*:
  - object descriptors and handles
  - direct memory pointers

## Handles (without swizzling)



# Handles

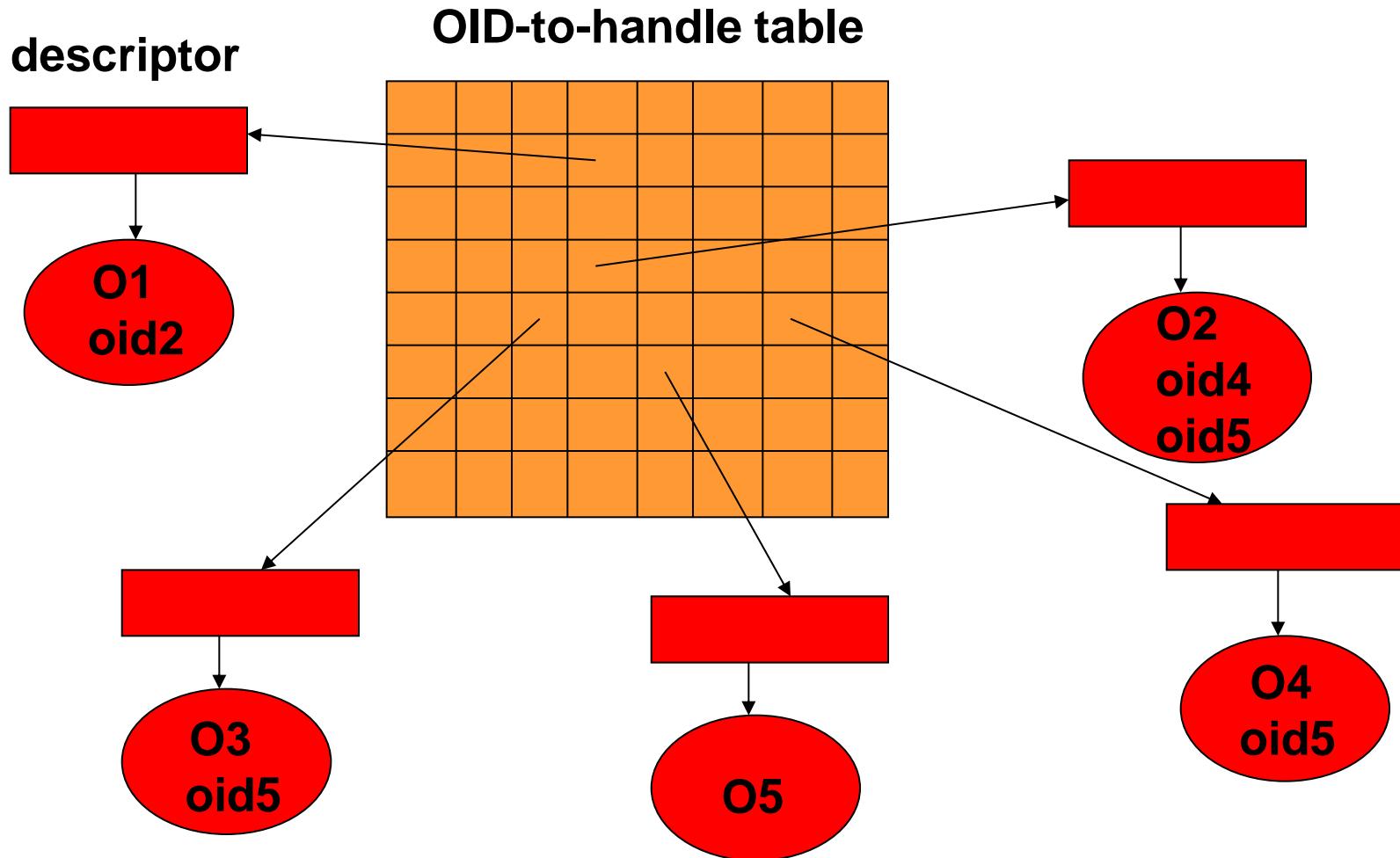


- **Handles** needed to keep *additional information* on objects (physical address, lock mode, etc.)
- Direct storage of this information in *OID-to-handle table* makes table too large (swapping)
- Handles simplify the *displacement* of objects from the address space of the application
- Hashing is costly (approx. 50 instr.) vs. direct addressing (1 instr.) → *eliminate hashing*
- Pointer *swizzling* replaces OID through pointer



# Object referencing without Pointer Swizzling

- Buffer management
- Record structure and addressing
- Access paths and indexing



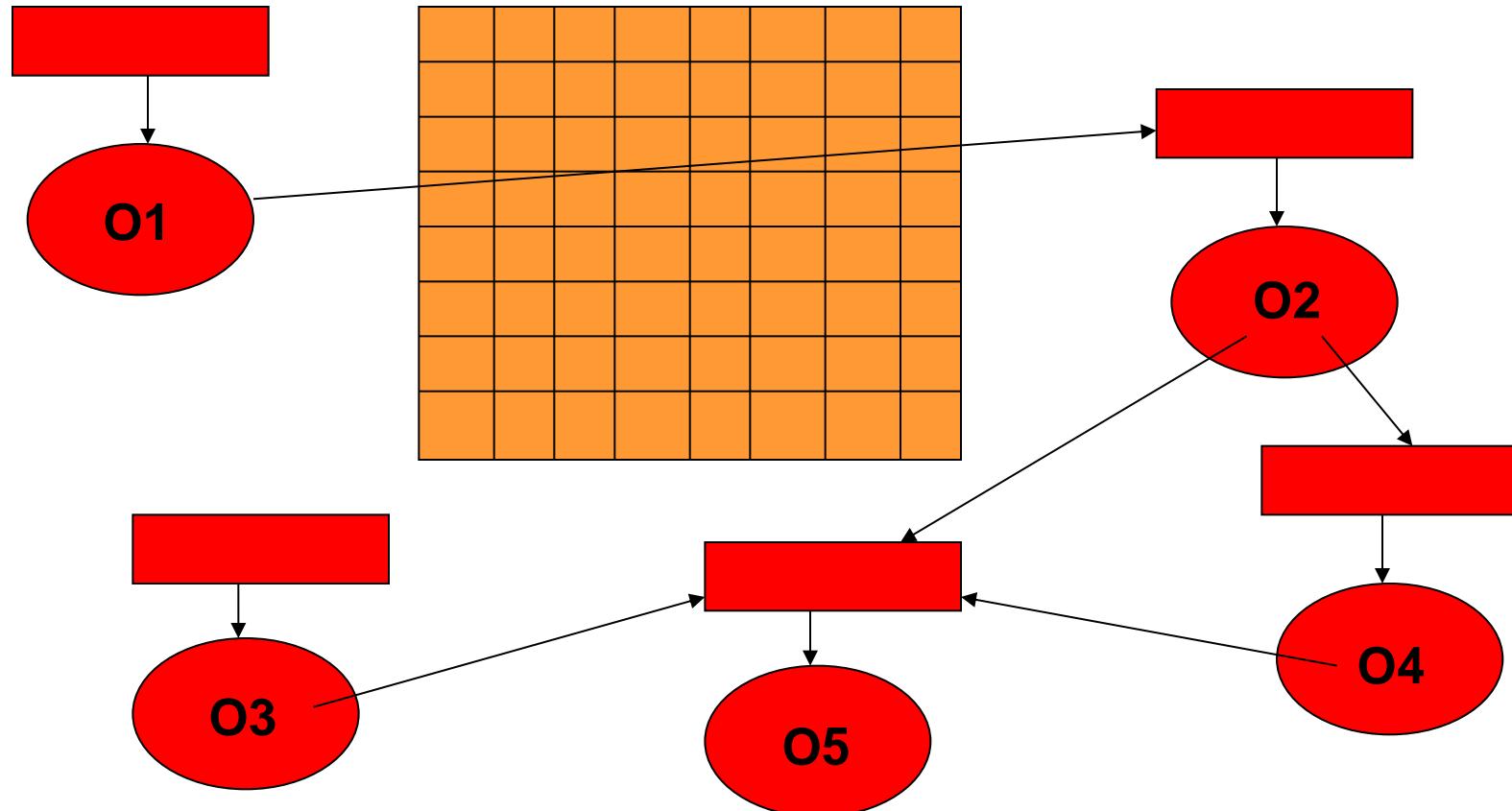
# Object Referencing after Pointer Swizzling

Buffer management

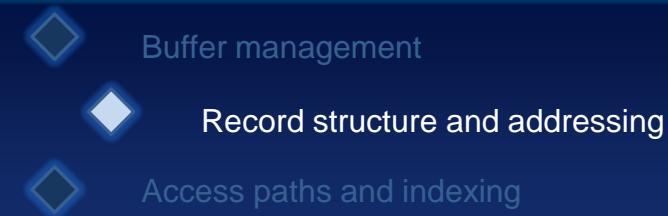
Record structure and addressing

Access paths and indexing

OID-to-handle table



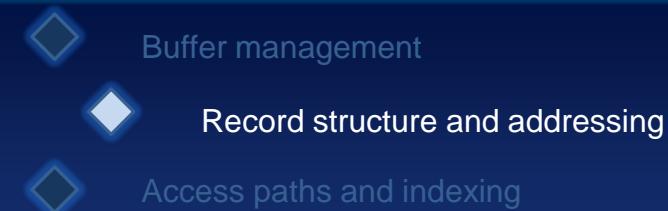
# Classification of Pointer Swizzling



- **In place** = on page in page buffer
- **copy** = copy object into object buffer
- **eager** = references swizzled up front
- **lazy** = references swizzled on demand
- **direct** = swizzle to physical address
- **indirect** = swizzle to handle/descriptor



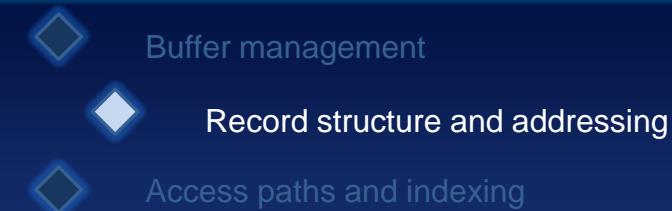
# Alternative Swizzling Classification



- **No swizzling**
  - no swizzling overhead
  - each reference requires access through hash T.
- **On first follow**
  - first reference through OID-to-handle table
  - replace OID through handle
  - next references use handle
  - at commit handles are replaced by OIDs
  - additional bits to distinguish between handle and OID



# Alternative Swizzling Classification



- **On first reference (Orion)**
  - first access to an object causes swizzling of all the references in that object
  - if referenced subobjects are not used, work was wasted
  - no need to distinguish between swizzled and unswizzled objects
- **Almost always (ObjectStore)**
  - swizzled on disk, very fast
  - upper bits of swizzled pointers repaired on fetch

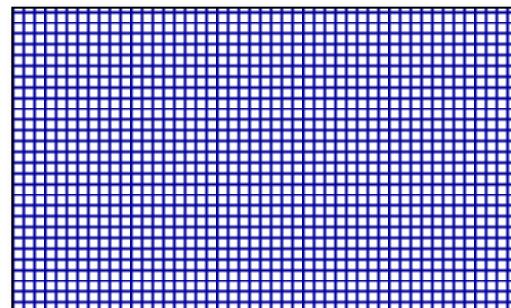
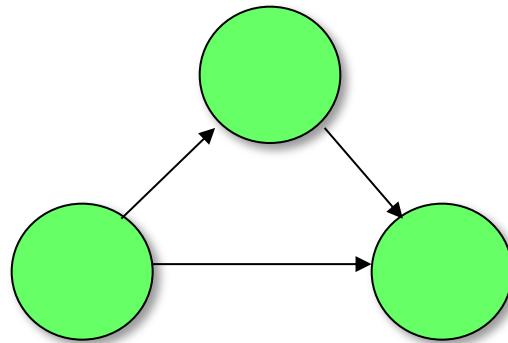


- Early **OODBMS** prototype, developed at MCC
  - object server
  - handles
  - swizzle on first reference
  - typed surrogate OIDs
  - format on disk different from format in main memory
- Interesting to analyze because of clear architectural choices

# Orion (contd.)

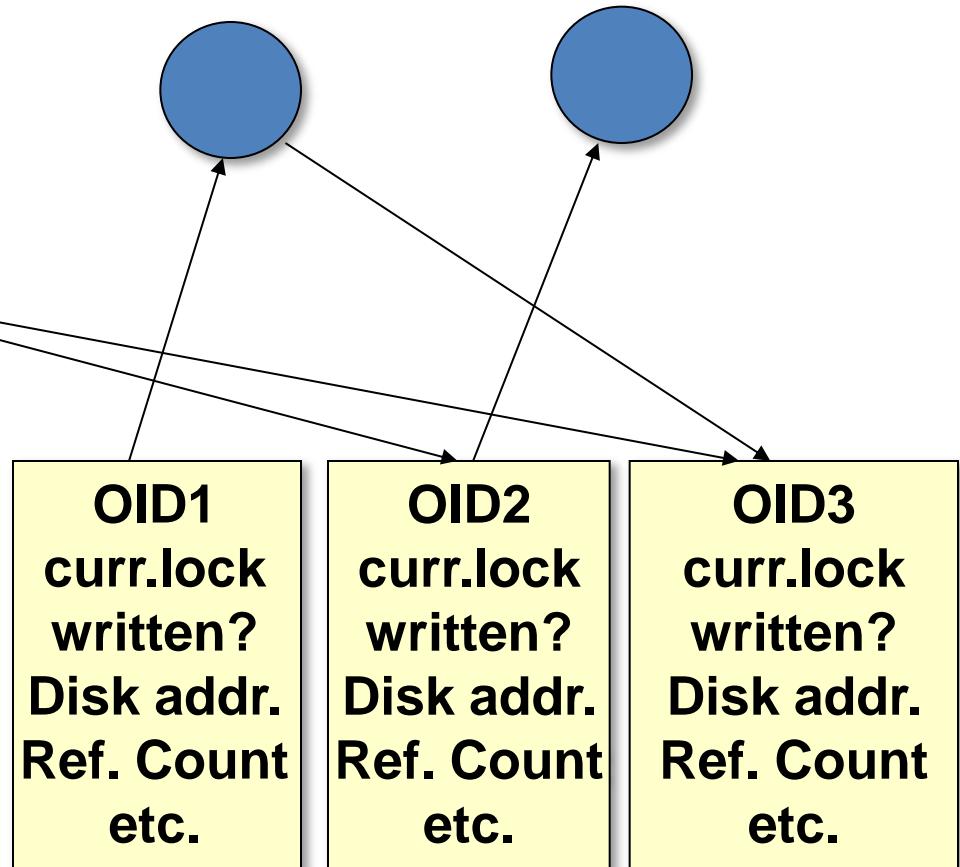
-  Buffer management
-  Record structure and addressing
-  Access paths and indexing

**Ordinary objects  
(application's heap)**

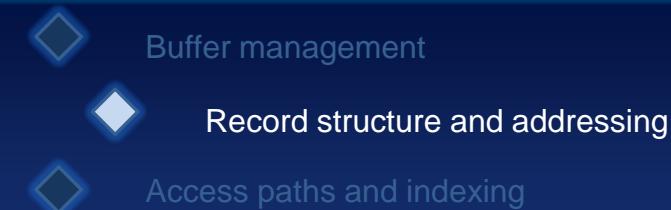


**OID-to-handle table for  
persistent objects**

**Persistent objects  
(DBMS buffers)**



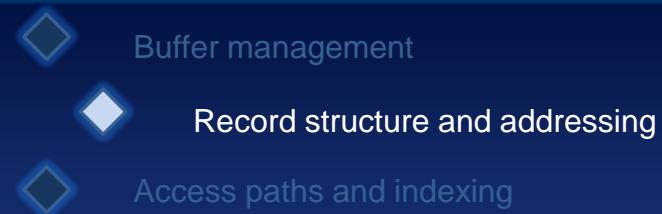
# Following a Reference in Orion



- Obj → next
  - control to DBMS
    - descriptor there → reference already swizzled
    - check locking mode (possibly request to upgrade)
    - if object already in memory, return ctrl. to appl. with object address
    - if object not in memory
      - (if necessary swap object, keep descriptor if ref. Count != 0)
      - get object from server, server converts format
      - allocate all descriptors for referenced objects, check with OID-to-handle table if already allocated, else allocate and replace OIDs through handles



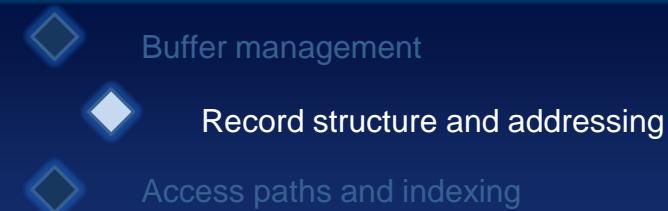
# Commit in Orion



- Write all modified objects (marked as dirty)
- Unswizzle references
- Send to server
- Server converts format, writes log, saves data



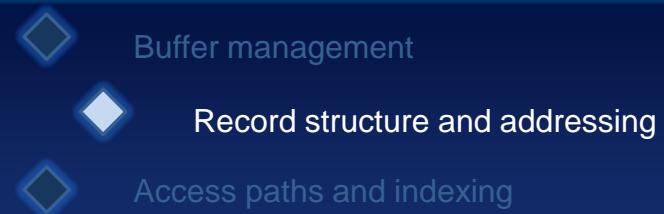
# Problems with Orion Architecture



- Large descriptors (percentage overhead significant when objects are small)
- Large descriptors consume memory and compete with application
- Paging of descriptors leads to *thrashing*
- Allocation/deallocation of descriptors slow
- Swizzle on first reference is *inefficient*



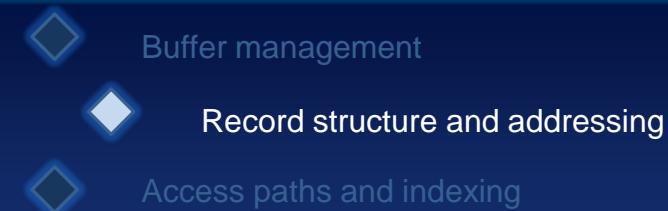
# ObjectStore



- Typical page server
- Buffers only pages on client side
- No handles or descriptors
- OIDs have physical address structure  
 $DB\# | seg\# | offset$
- “Almost always” swizzle



# Following a reference in ObjectStore



- Same structure in client buffers as in the persistent DB
- ObjectStore keeps mapping table of segments to addresses in virtual memory (Seg-to-Current-VM address table)

DB#	Segmt#	VMstart	VMend
1	1	100	249
1	2	400	499

Entered when O1 ref.

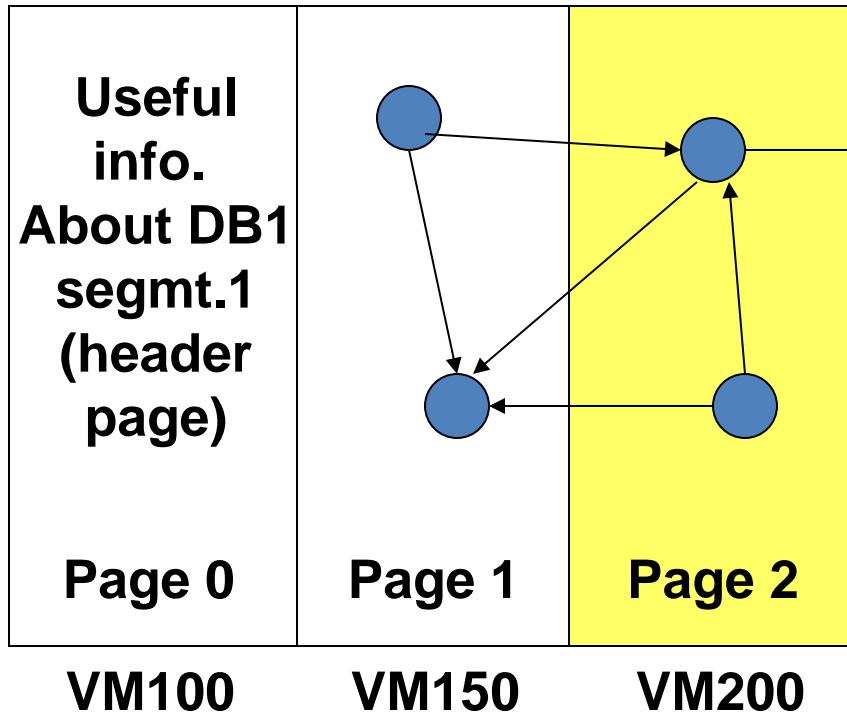
Entered when O2 fetched



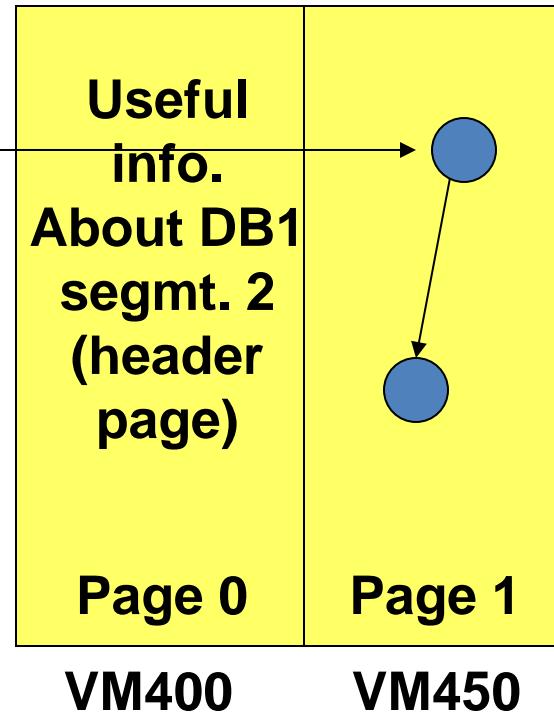
# Following Reference in ObjectStore

-  Buffer management
-  Record structure and addressing
-  Access paths and indexing

**DB1 segment 1**



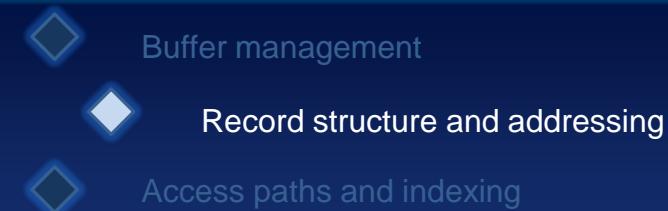
**DB1 segment 2**



**Shaded=in client's VM**



# Following a reference in ObjectStore



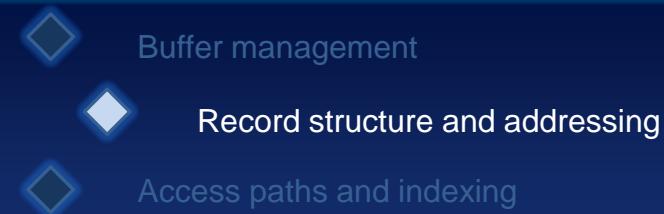
- **O1** references **O2** (lying on page 2 of segment 1)
- page-fault at VM200
- Operating System transfers control to ObjectStore
- ObjectStore locates page (DB1|segmt1|page2)
- Client requests from server (DB1|segmt1|page2)
- Server sets r-lock on page and transfers page
- Client must reformat page, consider offsets for pointers, provide space for referenced pages



# Need for reswizzling

- *OIDs/pointers* on disk correspond to canonical storage position (e.g. segmt1|page1 = VM0)
- If segment can be placed into canonical position in main memory it's ok, otherwise must add offset (VM of segmt start)
  - get beginning position from header page
  - for each object on page reswizzle address
  - pointers to other segments require allocation of new segment and changing pointer

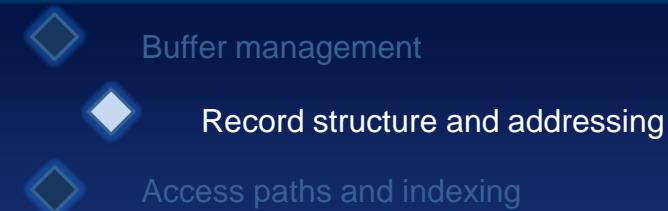
# Writing in ObjectStore



- Writing causes protection fault
- Operating system passes control to **ObjectStore**
- **ObjectStore** sets write lock
- Write-protection bit for page changed
- Write operation is now possible



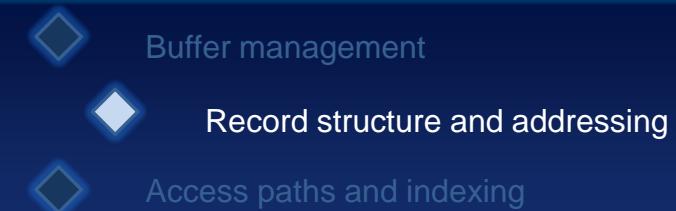
# Storage Overhead in ObjectStore



- Considerably less storage overhead than handles/descriptors
  - *segment-to-current-VM address table* - one entry per segment in each transaction
  - *object directory* - one per segment (independent of architecture)
  - *lock tables* - one entry per page
  - *OID-address conversion table* - one entry per reference to other segment



# Advantages/Disadvantages ObjectStore



- VM-Manager *only* passes control when page or protection fault occurs
- Page locks may cause *hot spots*
- *Clustering important* (swizzling efficiency, fewer intersegment references, fewer bottlenecks)
- VM must be *large enough* to hold referenced segments with all its data
- *Less storage requirement* than descriptor-based
- *Difficult to move objects on-line*, problem with growing/shrinking objects, migration in distr. Syst.



# Storage of Records

- Record manager must physically store records and provide operations to read, insert, modify and delete records
- Basic assumption: records do not span pages
  - $L_r \leq L_p - L_{ph}$
- Multiple records (possibly of different type) per page
- **Record structure** is stored in DB catalog
  - *name* (internal field name vs. external attribute name)
  - *length property* (fixed length, variable length, multiple)
  - *length* (number of Bytes)
  - *type* (alpha, numeric, binary, etc.)
  - *special storage properties* (cryptographic encoding, compression)
  - *treatment of special symbols* (NULL, leading 0s, blanks)



# Record Structure

**Goal:** record must be uniquely identifiable, storage efficiency, expandible while DB in operation, efficient calculation of a field's internal address

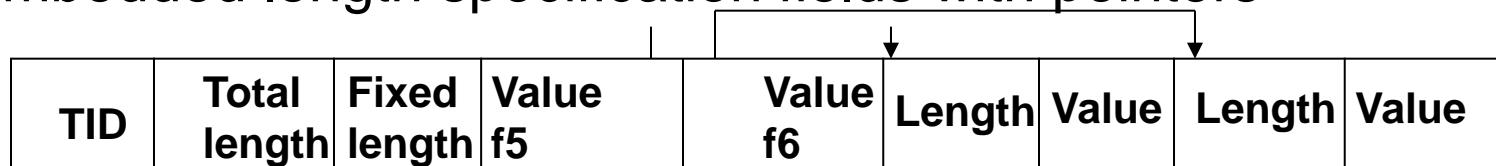
Record of type PERSON :PNR (f5), Name (v), Job (v), Salary (f6)

## Implementation options:

- fixed length, var fields allocated to maximum length
- embedded length specification

TID	Total length	Value f5	Length	Value	Length	Value	Value f6
-----	--------------	----------	--------	-------	--------	-------	----------

- embedded length specification fields with pointers



# Representation of Complex Objects

Buffer management

Record structure and addressing

Access paths and indexing

- Complex objects are constructed recursively out of basic constructors

```
complex_object STAFF [anchor_record_type =
staffanchor]
    set[...] of tuple (ID      [...] :      integer,
                      Name     [...] :      string(30),
                      Salary   [...] :      real,
                      CV       [...] :      var_string);
```

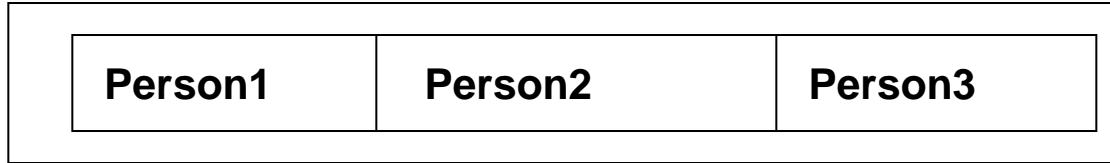
- Two orthogonal issues:**
  - implementation of the base constructors (sets, lists, tuples)
  - materialized storage vs. referenced storage for elements of sets and lists



# Storage of Complex Objects

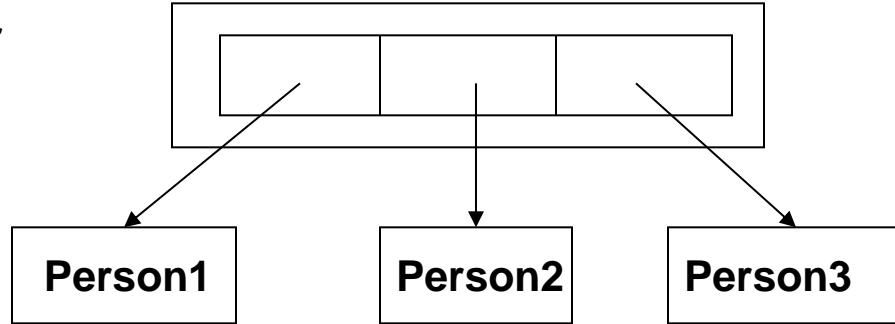
- ◆ Buffer management
- ◆ Record structure and addressing
- ◆ Access paths and indexing

- **Materialized storage:**



- **Referenced storage:**

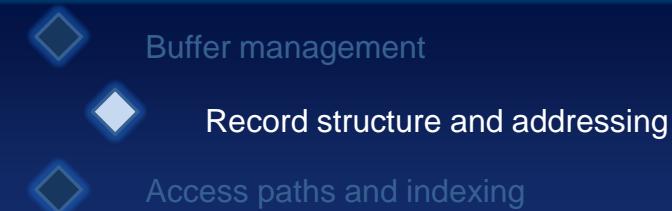
staffanchor



- Both options can be recursively combined resulting in many possible combinations



# Options for Storage of Complex Objects



- Different parameters determine storage of complex objects
- **Implementation type** (determines basic constructor)
  - variable length array
  - linked list
- **Placement type** (determines materialization of subobjects)
  - inplace (materialized)
  - referenced
- **Location type** (determines whether tuple is stored in one or more physical records, e.g. for long var\_strings)
  - primary
  - secondary



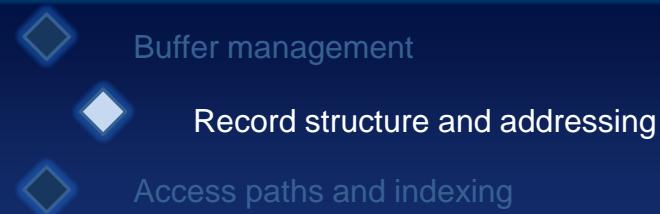
# Clustering

-  Buffer management
-  Record structure and addressing
-  Access paths and indexing

- **Clustering** tries to store logically related records in physical proximity
- Since disk is linear medium, *only one* clustering criterion can be enforced at a time
  - cluster according to one attribute in processing order
  - cluster all subobjects of a complex object
- Assign clustered objects to physically contiguous pages



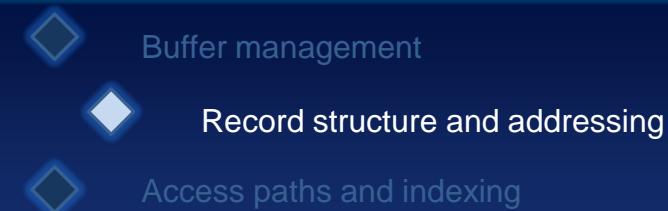
# Types of Pages in Informix US



- **Data pages** - contain the data rows for a table
- **Index pages** - root, branch and leaf pages of an index
- **Bit-map pages** - contain control information that monitors fullness of each page in the extent
- **Blobpages** - contain simple large objects that are stored with the data rows in the dbspace
- **Free pages** - pages that have been allocated but without specifying their use



# Informix US Structure of a Dbspace Page

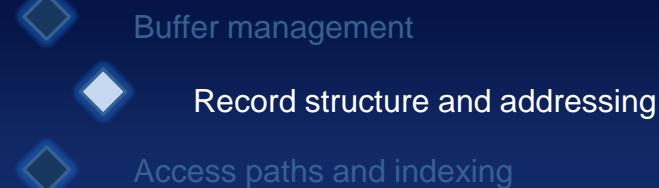


## Page consists of:

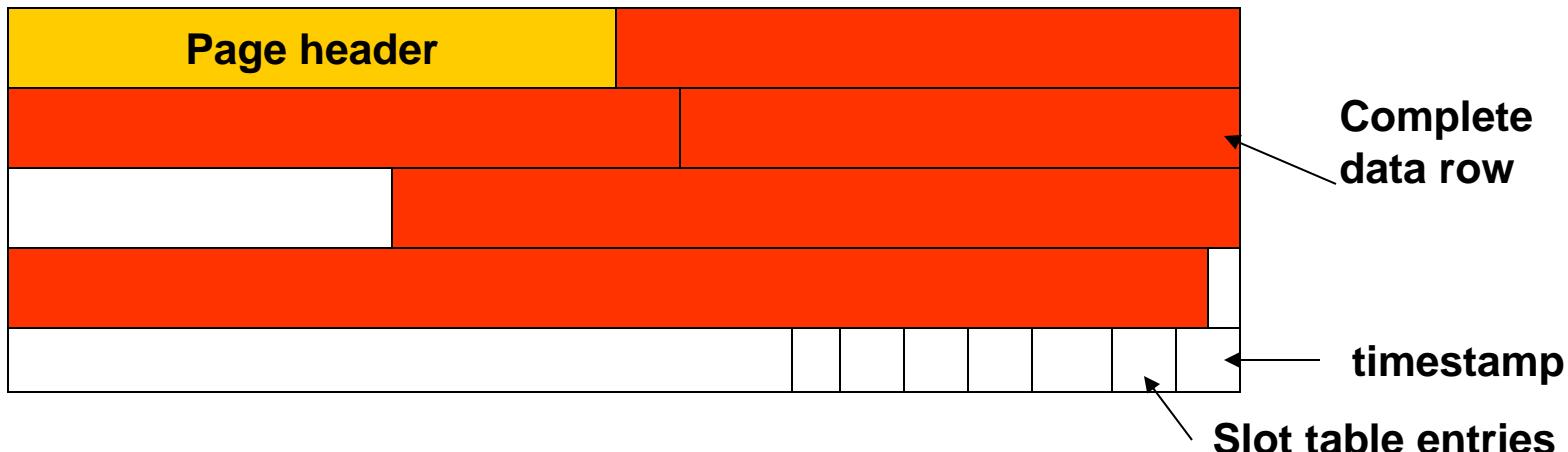
- **page header**
  - page ID
  - number of slot table entries
  - number of free bytes left on page
  - pointer to free space between last data entry and first slot
  - time stamp
  - two index-related pointers (used if page is index page)
- **page-ending time-stamp** (used for consistency w. header TS)
- **slot table** (up to 255 4-byte slot entries, grows from back to front)
  - slot table entry: page offset where data begins, length of data



# Rows, Rowids and Pages



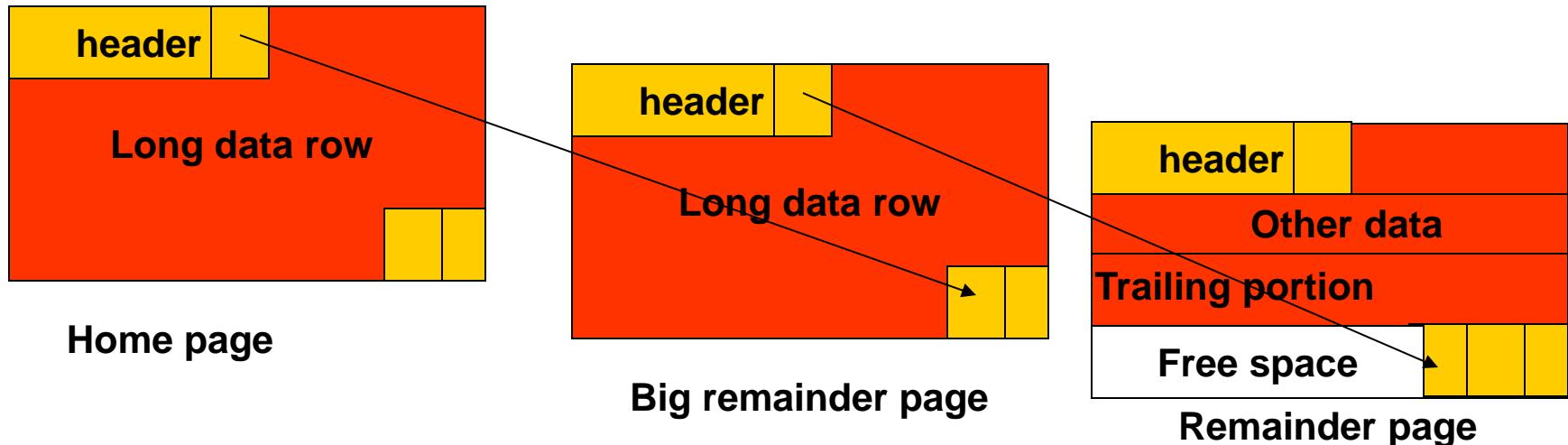
- **Rows** are identified in Informix US by **Rowids** which correspond exactly to the tuple-id concept (4 byte identifier of logical page-id and slot entry)
- **Rows** that are shorter than a **page** are always stored on one page
- **Page** is full if count of free bytes is less than max. row size



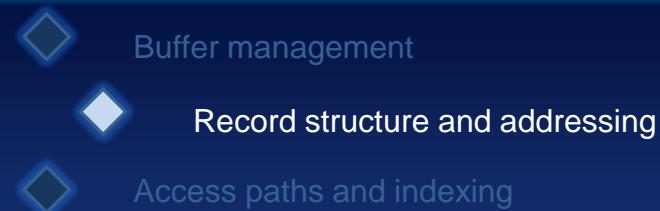
# Long Rows

- ◆ Buffer management
- ◆ Record structure and addressing
- ◆ Access paths and indexing

- **Rows** may be larger than a page
- **Rowid** for large rows is page-id of home page + offset
- Rest of the row is stored on remainder page: full remainder page = big-remainder page, trailing portion on separate remainder page



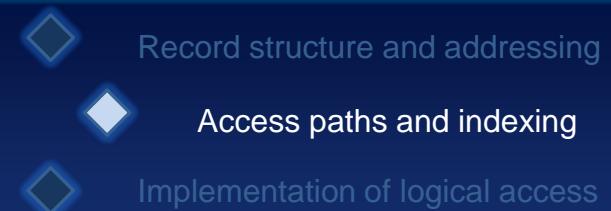
# Page Compression



- Free space on a page may get *fragmented*
- If free byte count is larger than required storage for a row but not enough contiguous space is available, the page is **compressed**
- During compression a page is copied to another buffer and the slot entries (offsets) are updated
- *After compression* page is copied back to original location so that Rowids don't change



# Access Paths



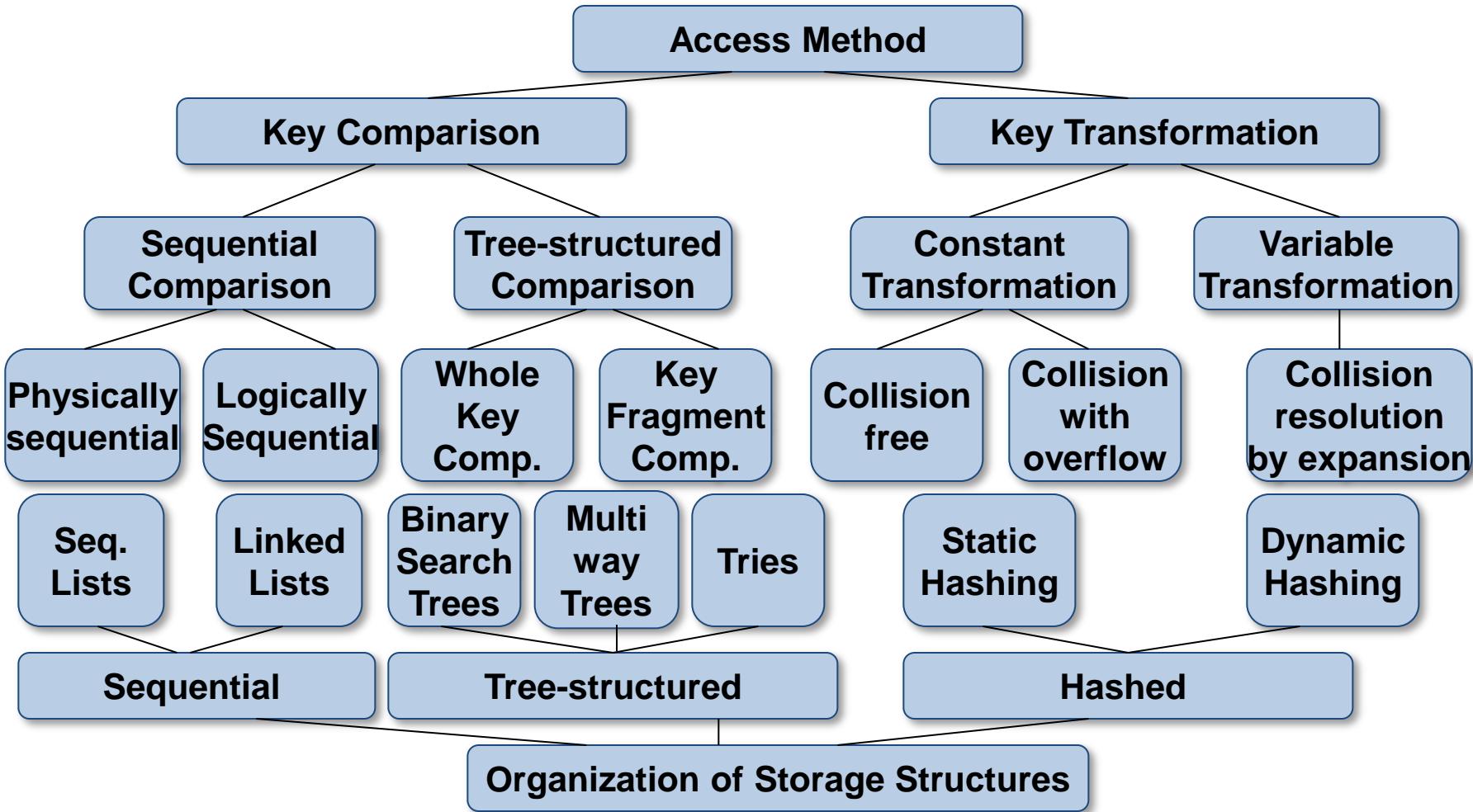
**Access paths** are needed to locate and retrieve records by key-value

- sequential access to all tuples of a table (scan)
- sequential access based on sorted attribute
- access to a single tuple via a primary key
- access to a set of tuples via a secondary key
- access to a (set of) tuple(s) based on composite keys and multidimensional searches (e.g. bit interleaved Z-order transform, other space filling curves)
- navigational access following references (pointers)



# Summary of One-dimensional Access Paths (Härder, Rahm)

- Record structure and addressing
- Access paths and indexing
- Implementation of logical access



# Multidimensional Access Paths

- Bitmap Indices
  - Simple, with and without compression
  - Encoded
- Space filling curves
  - Z-order encoding
  - Hilbert curves
  - UBtrees



# Comparison of Basic Access

Methods( $N = 10^6$  pg. acc., see  
Härder, Rahm for assumption)



Record structure and addressing



Access paths and indexing

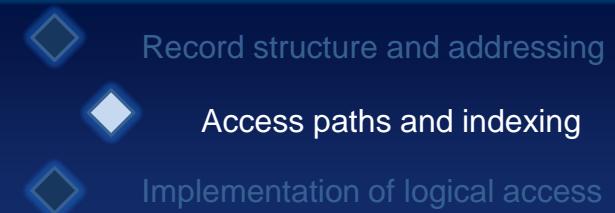


Implementation of logical access

Access Method	Storage Structure	Direct Access	Sequential A.	Blind Update
<b>Sequential Key Comparison</b>	<b>Sequential list</b> <b>Linked list</b>	$O(N) \approx 10^4$ $O(N) \approx 5 \times 10^5$	$O(N) \approx 2 \times 10^4$ $O(N) \approx 10^6$	$O(1) \leq 2$ $O(1) \leq 3$
<b>Tree-structured Comparison</b>	<b>Balanced binary tree</b> <b>Multiway tree</b>	$O(\log_2 N) \approx 20$ $O(\log_k N) \approx 3 - 4$	$O(N) \approx 10^6$ $O(N) \approx 10^6$	$O(1) = 2$ $O(1) = 2$
<b>Constant Key Transformation</b>	<b>External hashing with overflow</b> <b>External hashing with separators</b>	$O(1) \approx 1.1$ $O(1) = 1$	$O(N \log_2 N)$ $O(N \log_2 N)$	$O(1) \approx 1.1$ $O(1) = 1 (+ D)$
<b>Variable Key Transformation</b>	<b>Extensible hashing</b> <b>Linear hashing</b>	$O(1) = 2$ $O(1) > 1$	$O(N \log_2 N)$ $O(N \log_2 N)$	$O(1) \approx 1.1 (+R)$ $O(1) < 2$



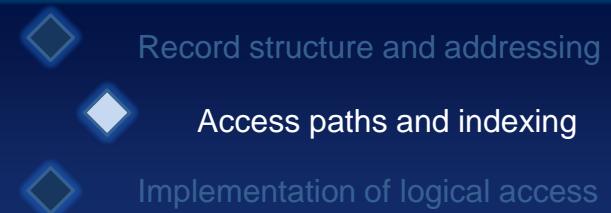
# Pointers to Access Path Information



- *Grundlagen der Informatik II (Skript)*
- Härder, Rahm:  
*Datenbanksysteme: Konzepte und Techniken der Implementierung*  
Springer
- Ramakrishnan:  
*Database Management Systems*  
McGraw Hill
- Any good book on data structures (e.g. Cormen)



# Informix US Indexing



*Two types of index offered*

- **B-tree (really B+ tree)**
  - entries consist of key value, rowid, delete flag
  - deletion causes only setting of flag, compression later
  - key shuffling to neighbor
  - pages are chained to neighbors at same level
  - bulk loading of index is possible
  - fill factor can be specified
- **R-tree**
  - used to index multidimensional geometric objects
  - R-tree operates on the principle of bounding boxes



Oracle offers multiple **index types**

- *B+ Trees*
- *Composite Indexes*
- *Function based Indexes*
- *Partitioned Indexes*
- *Index-Organized Tables*
- *Bitmap Indexes*
- *Bitmap Join Indexes*
- *Domain Indexes*
- *Clusters*
- *Hash Clusters*

# B-Trees and their use

- **SQL Engine** must maintain indexes even if these are not actively used
  - Index maintenance represents significant CPU and I/O resource demand in write-intensive applications
    - Do not build indexes unless necessary
    - Drop indexes that an application is not using
  - Determine index use through  
`ALTER INDEX MONITORING USAGE`
    - Monitors whether index has been used or not
    - Always monitor representative workload
- Indexes within an application can have non-obvious use
  - Foreign key index on parent table which prevents share lock from being taken on child table
- **SQLAccess Advisor** recommends alternative indexes to be created

# Choosing Columns/Expressions to Index

Record structure and addressing

Access paths and indexing

Implementation of logical access

- Consider *indexing keys* that are frequently used in WHERE clauses
- Consider *indexing keys* that are frequently used to join tables
- Choose *index keys* that have high selectivity
- *Do not* use standard B-tree indexes on keys with few distinct values (consider bitmaps)
- *Do not* index columns that are modified frequently
- Consider indexing foreign keys of referential integrity constraints when large number of concurrent  
INSERT, DELETE, UPDATE statements access the parent and child tables
- Consider whether the performance gain of queries offsets the performance penalty of updates, insertions and deletions



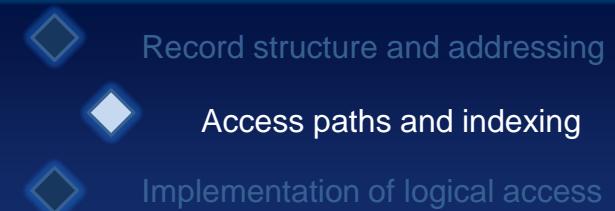
# Composite Indexes

- **Composite index** indexes two or more columns
  - Improved selectivity
  - Reduced I/O – if all columns selected by a query are in a composite index, query can be answered from the index without retrieving the base table
- *Construct* composite index

```
CREATE INDEX comp_ind  
ON table1 (x, y, z);
```

- Only leading portion can easily be used (x, xy, xyz)
- New option to use non-leading portion „Index Skip Scans“

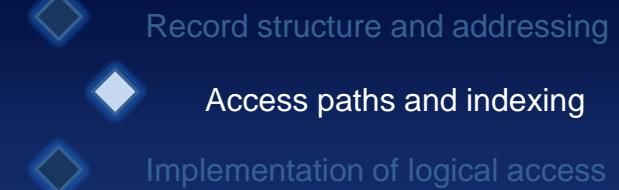
# Choosing Keys for Composite Indexes



- Keys that are used together in WHERE clauses combined with AND operators
- Keys in WHERE clause should make up leading portion
- Create the index in such key order that more frequently used attributes make up leading portion
- If all keys are used equally often in WHERE clause but data is ordered on one of the keys, place ordered key first



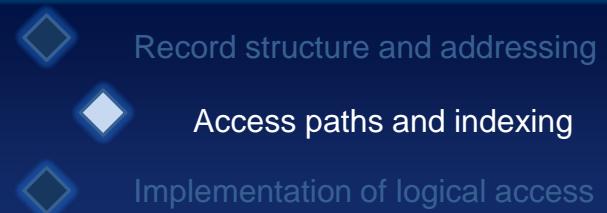
# Recreating and Compacting Indexes



- *Recreate* an index to minimize fragmentation
- Deletes do not automatically coalesce leaves/nodes
- After frequent DML activity index may be 50% full, if most columns of table are in index, then index might be larger than base table
- When creating a new index that is a subset of an existing index or when recreating an existing index, Oracle might use existing index instead of base table
- Leafs can be coalesced using `ALTER INDEX` with `COALESCE` option



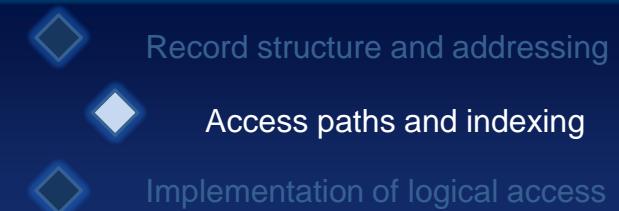
# Function based Indexes



- Function based indexes are indexes built on a transformation of columns or on expressions
  - `UPPER(last_name) = 'MARKSON'`
  - `col1 + col2`
- Computation intensive expressions can be stored in index



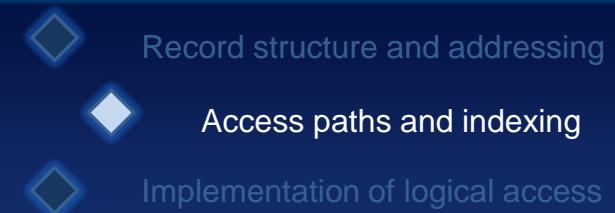
# Partitioned Indexes for Performance



- Tables can be partitioned based on some expression
- *Indexes can be partitioned same as tables*
  - Increased manageability, availability, scalability and performance
  - Range partitioning
  - Hash partitioning
    - Hash partitioning particularly useful when few index blocks suffer high contention (e.g. monotonically increasing keys inserted in B-tree)
    - Hash partitioning partitions entries to different partitions and spreads out contention



# Index Organized Tables for Performance



- Entire tuple is held in index
- Adding or deleting rows only results in index maintenance
- Very fast access for queries that involve exact match or range search or both



# Bitmaps for Performance

- **Bitmaps** are used whenever selectivity of keys is low (many tuples retrieved when searching for a key)
  - For very large tables
  - For tables that are relatively static (or append only)
- **Bitmaps** offer index cooperativity, i.e. queries can be answered solely accessing the index
- Particularly good for lengthy ad hoc WHERE clauses
- **Bitmaps** are good for join indexes
  - Bitmap join index stores corresponding rowids in separate table
  - Bitmap join indexes are more space efficient than materialized join views

# Bit-map indexing

$\text{IB}_a$	$\text{IB}_b$	$\text{IB}_c$	$\text{IB}_d$	T
1	0	0	0	...
0	1	0	0	a
0	0	1	0	b
0	0	0	1	c
1	0	0	0	d
0	0	0	1	a
				d

(Simple) Bitmap Index

- for example, select the suppliers whose Quality of Service is better than 5.0 and who supply some of the items —  $\{a, b, c\}$

```
SELECT *
FROM SUPPLIER S, SUPPLY SP
WHERE S.QOS > 5.0 AND
S.SID = SP.SID AND
SP.ITEM IN {a,b,c}
```



# Maintenance of simple Bit Maps

- Maintaining bit map indices is easy
- Two cases
  - no expansion of domain: add 1 in last position to vector corresponding to value of attribute in new tuple
  - with expansion of domain: add new vector with leading 0s and 1 in last position, add 0 to all other vectors
- Bitmaps vary in density (inversely proportional to cardinality of domain)
  - assuming even distribution, bitmaps are more efficient than TID lists for domains of 32 values or less



# Operations on simple bit map indices

- Bit map indices are scanned, position in bit vector corresponds to position of tuple in table

```
SELECT *
FROM SUPPLIER S, SUPPLY SP
WHERE S.QOS > 5.0 AND
      S.SID = SP.SID AND
      SP.ITEM IN {a,b,c}
```

- Selection predicate can be evaluated directly on bit-map index by ORing the bit vectors for SP.ITEM=a, SP.ITEM=b and SP.ITEM=c
- Bit-wise logical operations are fast



# Limitations of Simple Bitmap Indexing

- Simple bitmap indexing is unsuitable for large domains
  - increasing sparsity
  - performance degrades
- Solutions
  - compression of simple bitmaps (run-length encoding)
  - encoding the domain of the indexed attribute



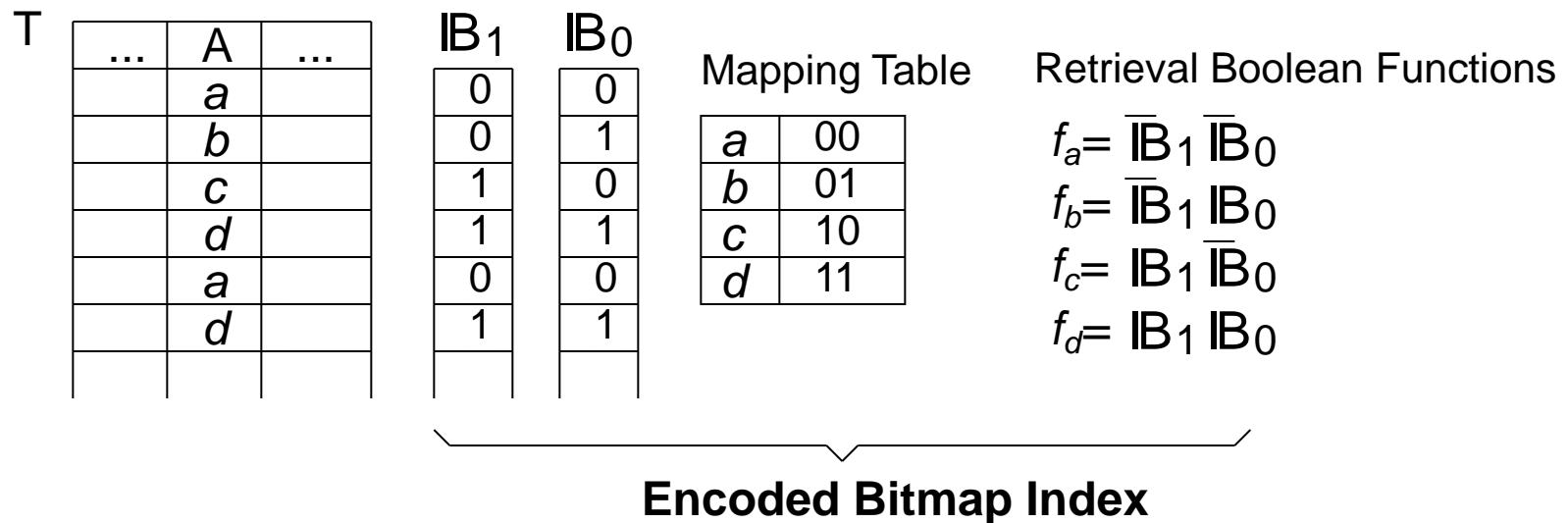
# Compression of simple bit maps

- Bit vector consists of sequences (of varying length) of 1 and 0
- Long runs of 0's or 1's can be encoded by substituting the repetitive run by an integer indicating how many 1's or 0's follow (run-length encoding)
- In B-tree each TID at leaf level takes typically 32 bits, no further compression possible
- In Bit maps  $n$  vectors (where  $n=\text{cardinality of domain}$ ) using 1 bit per tuple in each vector
- For small domains (less than 32 values) ==> few vectors, little compression possible
- For large domains ==> many vectors, more compression possible since longer homogeneous runs



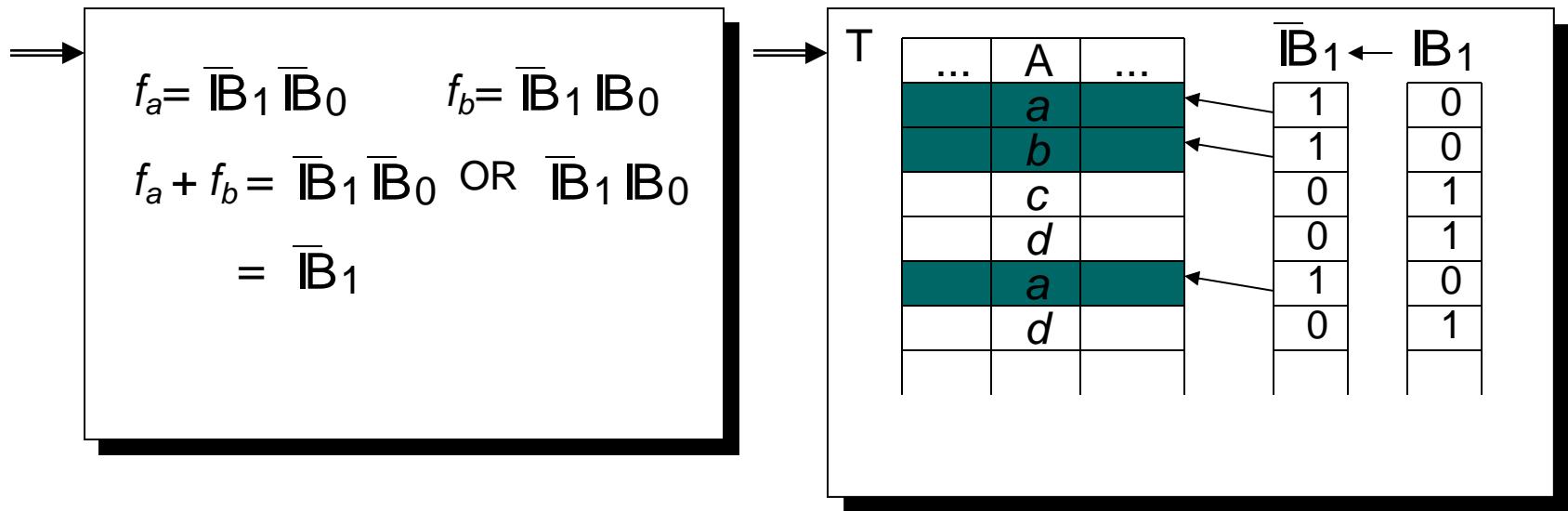
# Encoded Bitmap Indexing

- An encoded bitmap index consists of *a set of bitmap vectors (bit slices)*, *a mapping table* and *a set of retrieval Boolean functions*.



# How EBI works

```
SELECT *
FROM T
WHERE T.A in {a, b}
```



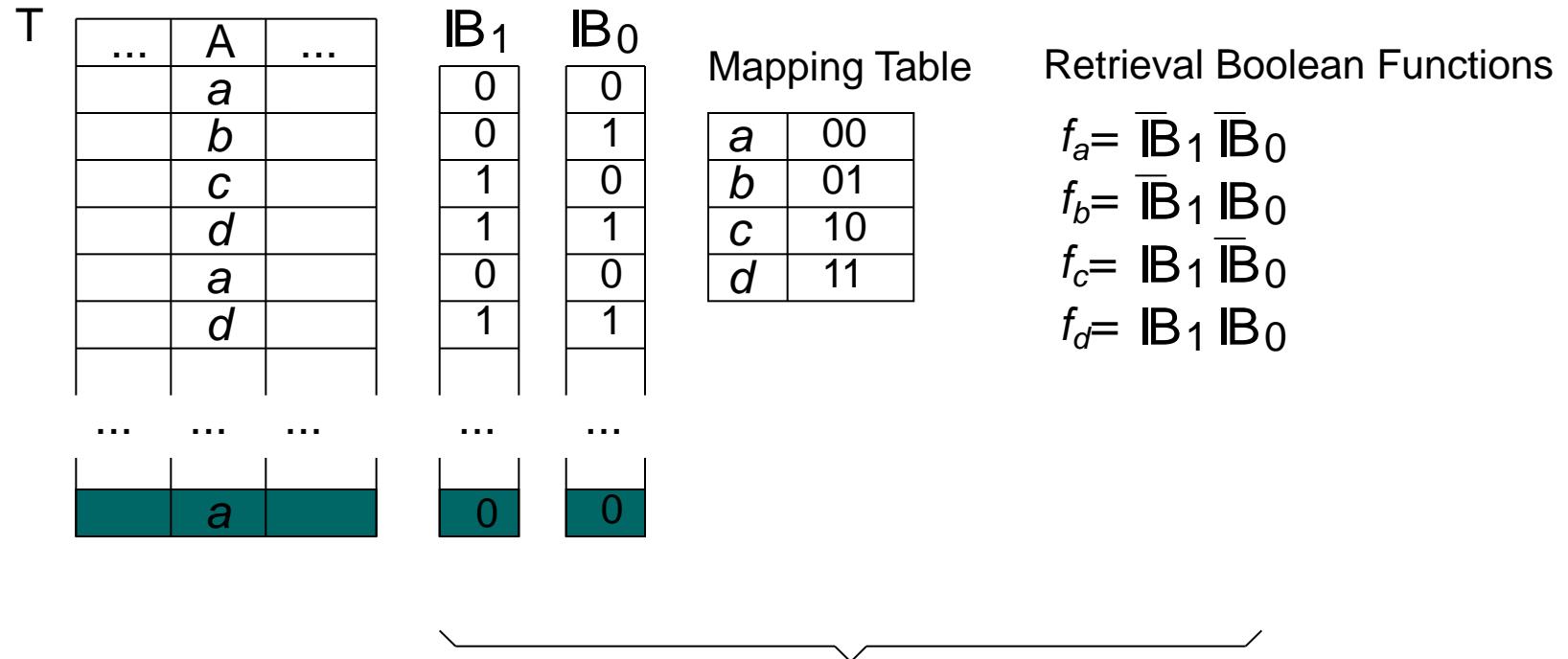
# Advantages of Encoded Bitmaps

- solve the sparsity problems arising from high cardinality
  - number of bit vectors =  $\log_2(\text{cardinality})$
- performance degradation is a **logarithmic** function of the cardinality of the indexed attribute
- cooperativity of different bitmaps
- provides more opportunities for (static/dynamic) optimization through logical reduction
- additional optimization possible through well-defined encodings



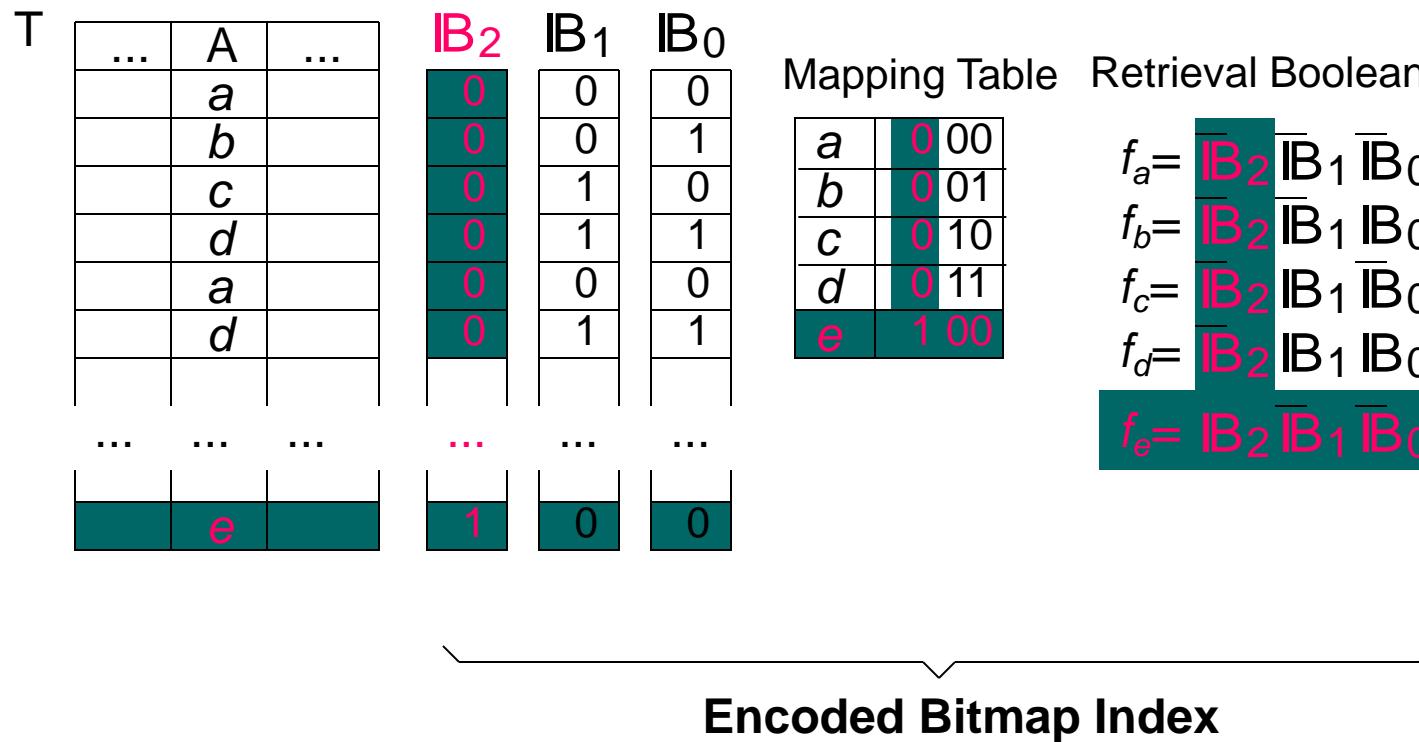
# Maintaining Encoded Bitmap Indices

- Maintenance due to updates
  - appending without domain expansion



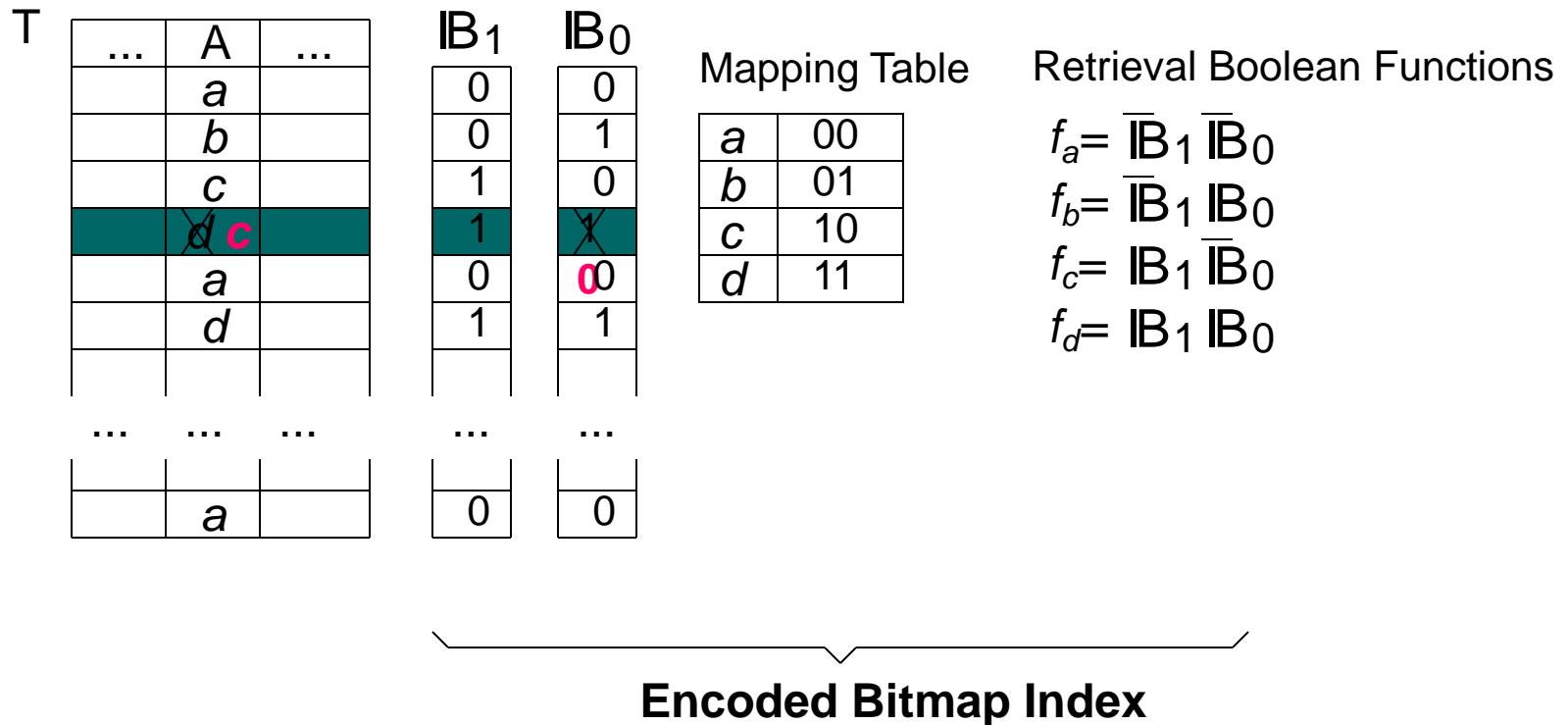
# Maintaining EBI

- appending with domain expansion

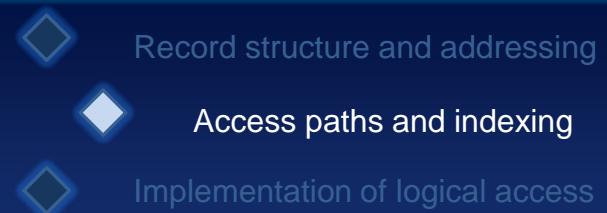


# Maintaining EBI

- other updates: deletions, updates
- NULL attribute values



# Domain Indexes

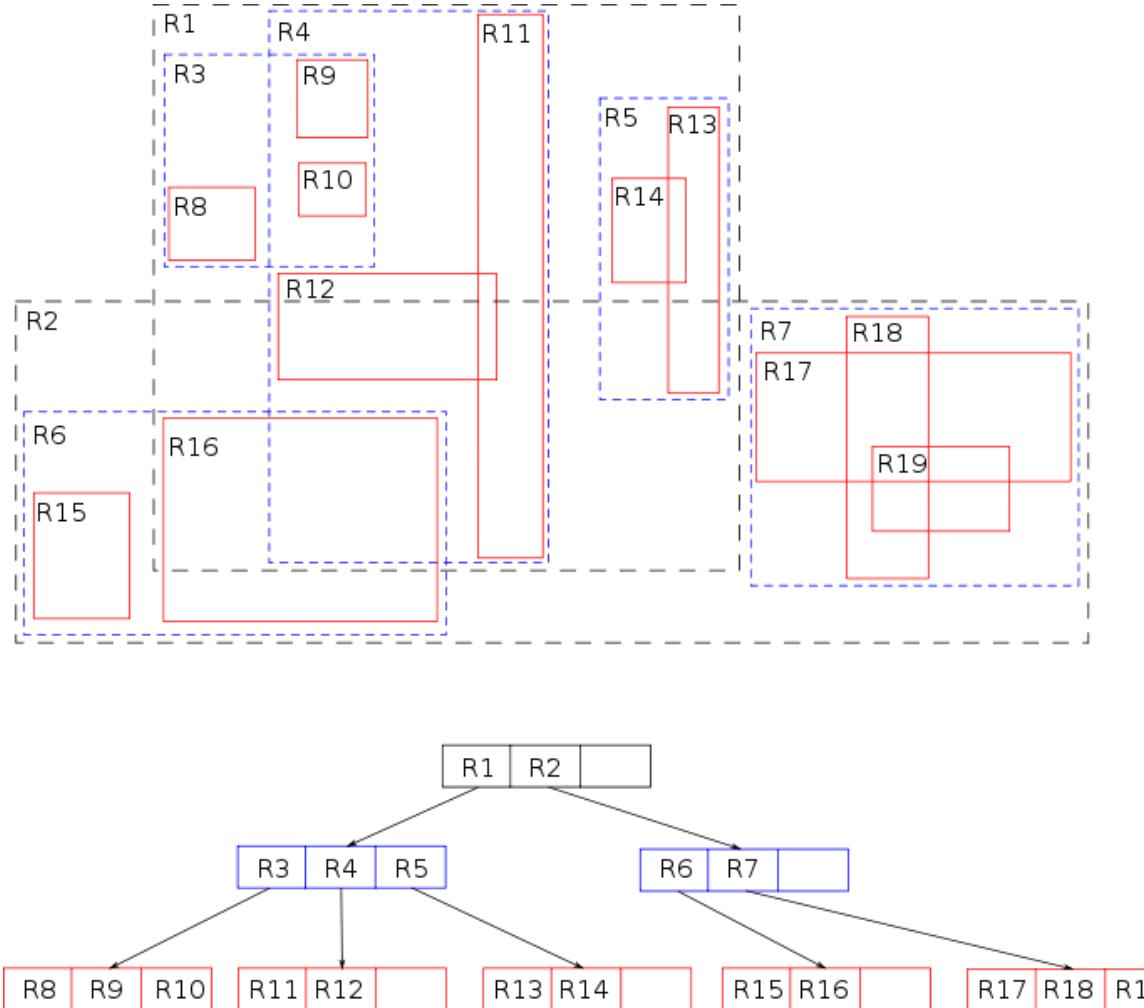


- **Domain indexes** are built using indexing logic supplied by a user-defined indextype
  - Spatial indextype
    - Bounding box (R-trees)
    - Space filling curve (Z-order encoding)
    - ...
  - Domain index types are provided through cartridges



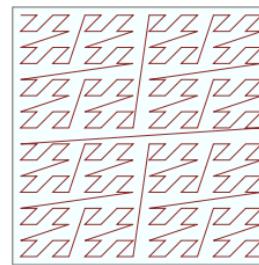
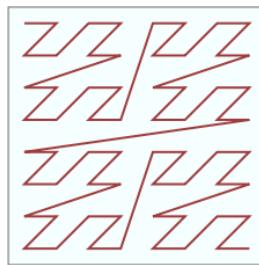
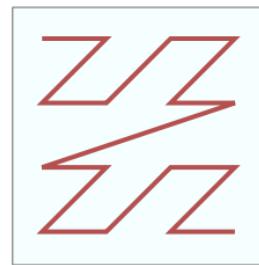
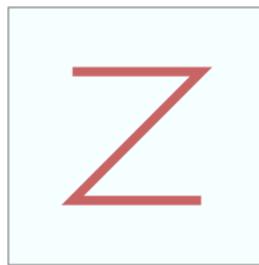
# R-Trees

- Based on minimum bounding box
- Multiway tree with similar properties as B-trees but lower fill (30-40% is best in 2D)
- Objects are grouped within rectangles
- If an upper rectangle doesn't intersect the query search predicate, none of contained objects will match



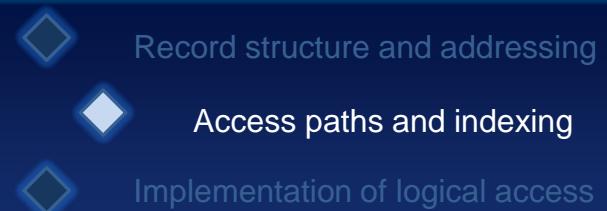
# Z-order encoding

- Z-order (a.k.a. Morton code) is a function that maps data from an n-dimensional space to one dimension while preserving locality.
- Z-value is calculated by bit interleaving of the binary representation of the coordinates (or attribute values in the various dimensions)
- Z-values can be stored/indexed in any index (e.g. B-tree)



x:	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
y:	0 000	000000 000001	000100 000101	010000 010001	010100 010101	010110 010111		
1 001	000010 000011	000110 000111	010010 010011	010110 010111				
2 010	001000 001001	001100 001101	011000 011001	011100 011101				
3 011	001010 001011	001110 001111	011010 011011	011110 011111				
4 100	100000 100001	100100 100101	110000 110001	110100 110101				
5 101	100010 100011	100110 100111	110010 110011	110110 110111				
6 110	101000 101001	101100 101101	111000 111001	111100 111101				
7 111	101010 101011	101110 101111	111010 111011	111110 111111				

# Clusters



- Oracle allows storing rows of different tables that are used together on same page
- **Cluster tuples together** if they are
  - accessed frequently by application in join statements
  - master-detail records often accessed together
- **Do not cluster tables** together if
  - application performs full table scan of only one table
  - the clustering attribute is often modified (update requires physical moving of record)
  - all tuples corresponding to a cluster value occupy several blocks (Oracle reads all blocks for a single row access)
  - cardinality of values is very different (some key values have many tuples some very few)



# Logical Access Paths and Logical Data Structures

Access paths and indexing

Implementation of logical access

Query optimization

Set-oriented Interface

Query Languages SQL, QBE, OQL

Relations, Views, Tuples, Objects

Transactional programs

Addressing units: relations, views, tuples

Auxiliary structures: external schema descr., integrity constraints

Addressing units: external records, sets, keys, logical access paths

Logical Data Structures

Addressing units: external records, sets, keys, logical access paths

Auxiliary structures: access paths, internal schema description

Addressing units: internal records, B-trees, hash-tables, etc.

Logical Access Paths

Internal interface



# Schema Description - The Catalog

Access paths and indexing

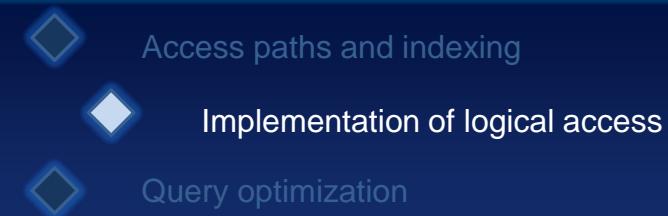
Implementation of logical access

Query optimization

- **The catalog** (also called Data Dictionary/Directory, Repository) contains the metadata that describe
  - *conceptual schema* (names, structures, constraints, access rights)
  - *internal schema* (storage structures, encoding, size, physical storage information)
  - *compiled views, stored procedures*, etc.
  - *statistics* (cardinality, selectivity, etc.)
- (Parts of) the catalog often stored as relations and accessible via the query language (SQL)



# Record Oriented vs. Set Oriented Access



- Typical **record oriented accesses** are
  - INSERT
  - UPDATE
  - DELETE
  - single tuple access via primary key
- Typical **set oriented accesses** are
  - range queries
  - aggregation operations
  - any query returning a set of tuples



# Implementation of Operations

(Recommended reading: Ramakrishnan)

Access paths and indexing

Implementation of logical access

Query optimization

- Need to understand how each logical operation is implemented
- **Each logical operation** may have multiple implementations depending on
  - *available access paths*
  - *sorting order*
  - *type of query* (single value, range query, etc.)
  - *subsequent processing* (individual operator vs. iterators and pipelining)
  - *available buffer and buffer management policy*

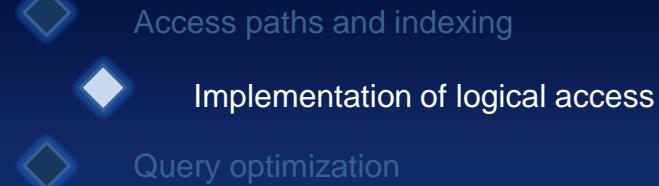


# Relational Operations

- **Unary operators**
  - selection
  - projection
- **Binary operators**
  - join
  - cross product
  - intersection
  - union
  - set difference
- **Grouping operators**
  - aggregate operators
  - group by



# Selection



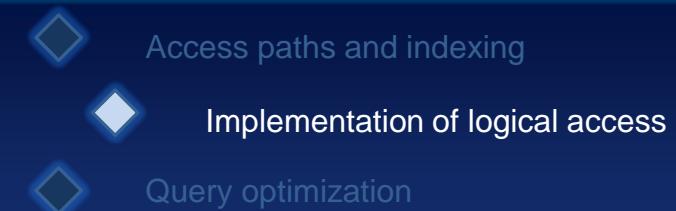
- Will use simple selections of the type

$$\sigma_{R.\text{attr} \theta \text{ value}}(R)$$

- Relation R has M pages
- Unsorted data, no index
  - only possibility is to scan the relation
  - I/O cost = M accesses
  - compare each tuple and add to result set if match
  - I/O cost of writing result depends on selectivity



# Selection - No Index, Sorted Data



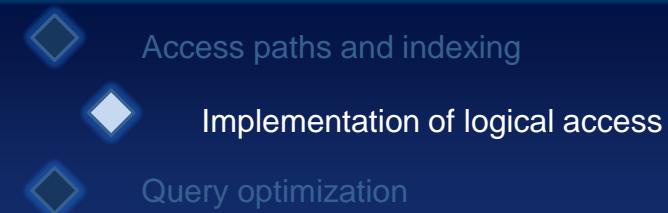
- If relation is sorted on selection attribute, use sort order to do binary search
- Continue sequentially until selection predicate no longer satisfied
- Cost of binary search:  $O(\log_2 M)$
- Cost of record scan after locating entry point: proportional to number of tuples satisfying search criterion (may vary from 0 to M I/Os – notice difference between #tuples and #pages=I/Os)
- Cost of writing result set
- In practice unlikely that relation will be kept sorted (but intermediate results might be)



# Selection with B+ Tree Index

- **Equality query vs. Range query**
  - B+ index particularly effective for range queries
  - for equality it is ok, but hash would be better
- **Search** tree to find first entry matching search predicate (2 - 3 accesses or more depending on size of relation)
- **Scan** leaf pages for all entries satisfying search criterion (function of selection predicate)
- **Retrieve** for each entry the corresponding tuple in R
  - big difference whether index is clustered or not
  - difference may be factor 50 (Härder & Rahm)
- Use of unclustered B+ tree for range query is inefficient  
→ scan whole relation

# Selection with general selection conditions



- General selection condition: terms of the form  
**R.attr  $\theta$  value**  
**R.attr  $\theta$  R.attr**  
**connected by  $\wedge$  or  $\vee$**
- Need to normalize this representation for parallelization purposes



# Query Normalization

- WHERE clause may have arbitrary complexity
  - quantifier-free predicate
  - preceded by all necessary quantifiers ( $\forall, \exists$ )
- Two possible normal forms:
  - conjunctive normal form (conjunction of disjunctions)

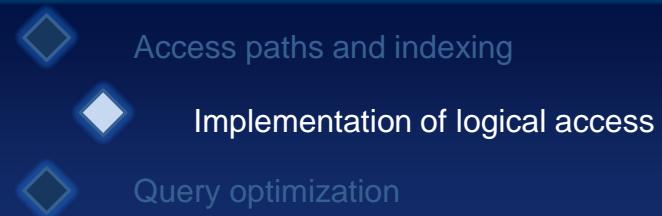
$$(p_1 \vee p_2 \vee p_3) \wedge \dots \wedge (p_{n-3} \vee p_{n-2} \vee p_{n-1})$$

- disjunctive normal form (disjunction of conjunctions)

$$(p_1 \wedge p_2 \wedge p_3) \vee \dots \vee (p_{n-3} \wedge p_{n-2} \wedge p_{n-1})$$



# DFN vs. CFN



- In disjunctive normal form the query can be processed as independent conjunctive subqueries linked by unions
- **DNF** may lead to replicated join and select predicates because in the original predicates the individual predicates are more often linked by conjunctions
- **CNF** is more practical when predicates involve more ANDs than ORs (the normal case)



# Selection without Disjunction

**Two options when no conjunct contains a disjunction:**

- Retrieve tuples using a *file scan*
- Retrieve tuples using an *index* on the primary term and compare each tuple on the non-primary terms
  - number of tuples retrieved depends on the selectivity of the primary terms
  - secondary terms only influence the cardinality of the result set
- *Utilize several indexes*
  - apply each available index independently and produce sets of tids
  - intersect sets of tids
  - apply secondary selection predicates to intermediate result set



# Example of Selection without Disjunction

Access paths and indexing

Implementation of logical access

Query optimization

## Sample DB schema

*Sailors* (sid:int, sname:string, rating:int, age:real)

*Reserves* (sid:int, bid:int, day:date, rname:string)

Predicate  $rname = 'Joe' \wedge bid = 5 \wedge sid = 3$

- Retrieve exact match with composite index on rname,bid,sid
- Retrieve match on rname = 'Joe' with B+Tree on rname, compare other fields on retrieved tuples

Predicate  $day < 8/9/94 \wedge bid = 5 \wedge sid = 3$

- retrieve tid-list using B+ tree on day
- retrieve tid-list using hash index on sid
- intersect both sets of tids
- evaluate bid = 5 on intermediate result set

# Selection with Disjunction

- Consider the **selection predicate**

$$day < 8/9/94 \vee bid = 5$$

- if either one requires a full scan, the whole relation must be scanned
  - even if an index exists on bid, the disjunction forces us to analyze all tuples because of the predicate on day
- Consider now the **predicate**

$$(day < 8/9/94 \vee bid = 5) \wedge sid = 3$$

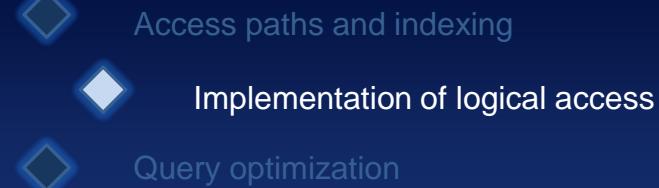
- if an index exists on sid, only those tuples must be evaluated

# Elements of a Cost Function

- **Access cost to secondary storage**
  - cost of accessing and transferring data blocks from disk to main memory
  - note that accessing one or many tuples within a block carries the same I/O cost
- **Storage cost**
  - cost of storing intermediate results
- **Computation cost**
  - cost of performing operations on data in memory (searching, sorting, merging, performing computations of field values)
- **Communication cost**
  - cost of shipping queries and results between client and server



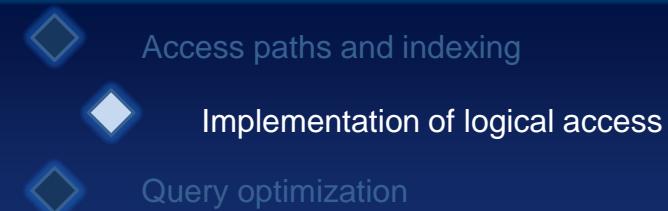
# Cost Functions



- What costs should be considered in a cost function?
- **Depends on situation:**
  - *very large databases* - I/O costs (including costs of writing intermediate results) dominate
  - *small databases* that fit completely in main memory - computation costs dominate
  - *distributed databases* - communication costs dominate
- Difficult to produce good weighted cost functions
- Will concentrate on I/O costs knowing that computation costs are significant



# Information Used in Cost Functions



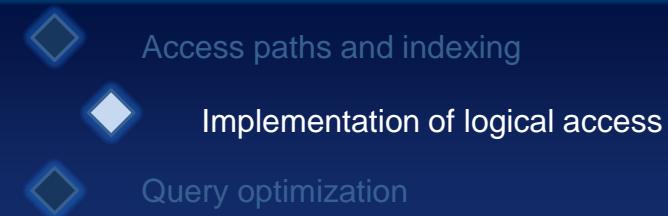
- **Statistics on each file**
  - total size of file in blocks -  $b$ , size of record, blocking factor  $bf$  (# of records that fit in a block)
  - primary access path available (ordering, type of index - hash, clustered, non-clustered)
  - secondary indexes
  - number of levels of an index and/or number of first level index blocks, highest/lowest key
  - number of distinct values of an attribute influences the selection cardinality
    - $s = r/d$  where  $r$  is the # of tuples,  $d$  is # of distinct values
    - for keys  $s = 1$
- Optimizer will need „reasonably“ good statistics (trade-off)



# Cost Functions for Selection

- Strategy: **linear search** (brute force)
  - retrieve all blocks of file:  $C = b$
  - for equality constraint on key:  $C = b/2$
- **Binary search**
  - for arbitrary value:  $C = \log_2 b + \lceil(s/bf) \rceil - 1$
  - for key attribute:  $C = \log_2 b$
- **Index access**
  - tree index requires index level + 1:  $C = x + 1$
  - hash:  $C \approx 1$

# Cost Functions for Selection (cont.)



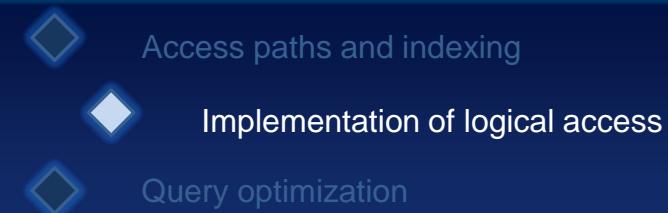
- Using **ordering index** in range queries (comparisons involving  $>$ ,  $<$ ,  $\leq$ ,  $\geq$ ) :  $C = x + b/2$   
Note that this is very crude estimate that may be right on average but may be quite wrong for individual case
- Using **clustering index** to retrieve multiple records (equality condition on attribute w. selection cardinality  $s$ ):  
 $C = x + \lceil (s/bf) \rceil - 1$
- Using **non-clustering B+ tree**:
  - non-key attributes:  $C = x + s$
  - key attribute:  $C = x + 1$



# Projection

- **Projection** is simple if key is part of project-list
  - result set has same number of tuples as original relation
  - fewer attributes per tuple
- If key attribute(s) not part of project-list
  - result set may contain duplicates
  - proper projection eliminates duplicates
  - eliminate duplicates by
    - **sorting** - equal values will be adjacent, eliminate them by sequential comparison
    - **hashing** - equal values will hash to same bucket, compare value of hashed key against other values in bucket and eliminate duplicates

# Sorting vs. Hashing for Projection



- **Sorting** is superior to **hashing** if many duplicates exist or if distribution is non-uniform
- *Useful side-effect of sorting:* result is in sorted order (which is often required for final result)
- **In general:** sorted results can be pipelined
  - to user
  - to next step in processing (e.g. as input to join algorithm)



# Cost Function for Projection

- The **cost for (naive) projection with duplicate elimination** will be:

$$C = b + t + at(\log t)$$

one full scan, cost of writing temporary relation, cost of sorting, a is a constant

- Can improve through combination of reduction and first pass of external sorting algorithm, and by eliminating duplicates during merge process

$$C = b + 2t$$



# The Join Operation

- Consider key-foreign key join query:

*SELECT \**

*FROM Department D, Employee E*

*WHERE D.eno = E.eno*

- Join can be thought of as cross product followed by selections and projections, however direct implementation much more efficient
- Two basic strategies:**
  - underlying enumeration of cross product
  - partitioning without enumeration of cross product

# Nested Loop Join

- Enumerates the underlying cross product and discards non-matching tuples

```
for each tuple r ∈ R do
    for each tuple s ∈ S do
        if ri == sj then add ⟨r, s⟩ to
result
```

- Scan the outer relation  $R$  and for each tuple  $r \in R$  scan entire inner relation  $S$
- This should be done page or block at a time, i.e. if  $R$  has  $br$  blocks and  $S$  has  $bs$  blocks

$$C = br + br * bs + brs$$

# Block Nested Loop Join

- Previous discussion did not consider buffers
- **Special case:** suppose there is enough buffer to hold the entire smaller relation (e.g. R) with two extra pages
- We can load the entire smaller relation and use one extra page for the bigger relation and one to write results

$$C = br + bs + brs$$

- Each relation is scanned once and  $brs$  is the cost for writing the result set
- **Generalizing:** buffer use should be adjusted to available buffer, important which relation is inner or outer
- Smaller relation should be used as outer relation if it does not fit in buffer

# Example Nested Loop Join

Relation **Employee** consists of  $r_E = 5000$  records stored in  $b_E = 2000$  disk blocks

Relation **Department** consists of  $r_D = 50$  records stored in  $b_D = 10$  disk blocks

There are  $n_b = 7$  buffers available, one reserved for writing result  
Using Employee as outer relation:

Total # of blocks accessed for outer relation =  $b_E$

# of times  $(n_b - 2)$  blocks of outer file are loaded =  $\lceil b_E / (n_b - 2) \rceil$

Total # of blocks accessed for inner rel. =  $b_D * \lceil b_E / (n_b - 2) \rceil$

# Example Nested Loop Join (cont.)

- Access paths and indexing
- Implementation of logical access
- Query optimization

**Total block accesses:**

$$b_E + (\lceil b_E / (n_b - 2) \rceil * b_D) = 2000 + (\lceil 2000 / 5 \rceil * 10) = 6000 \text{ block accesses}$$

Using Department in outer relation

$$b_D + (\lceil b_D / (n_b - 2) \rceil * b_E) = 10 + (\lceil 10 / 5 \rceil * 2000) = 4010 \text{ block accesses}$$

Generalized cost function?



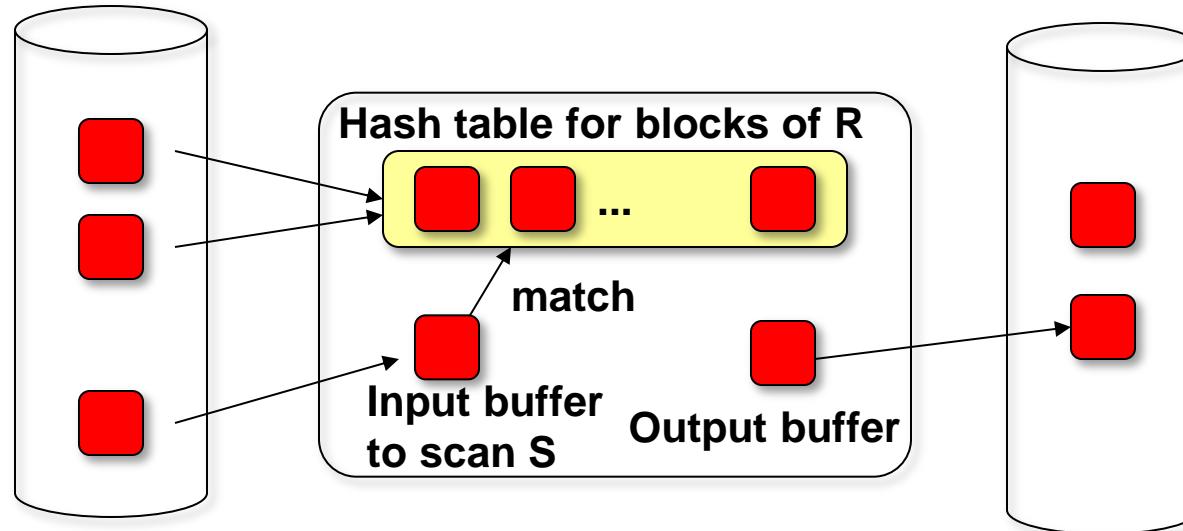
# Using Hash in Nested Loop Join

Access paths and indexing

Implementation of logical access

Query optimization

- When loading a block we may build a hash table on loaded block
- **Hash table** takes little extra space and speeds up match



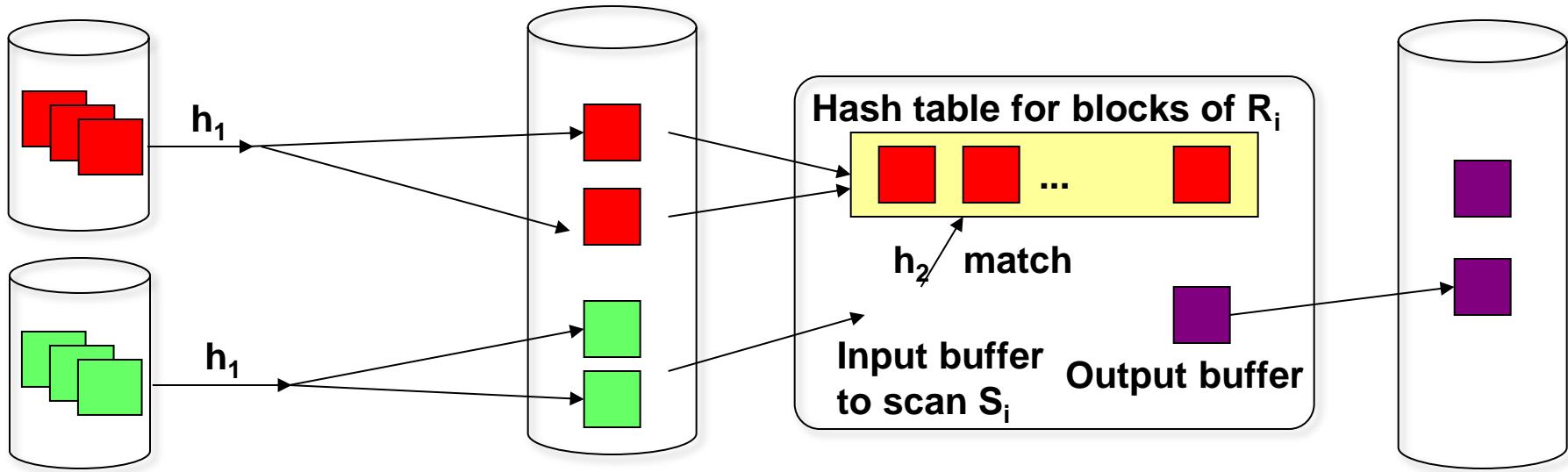
# Hash Join

- Hash join identifies partitions based on hashing and then compares tuples (for equality) within partitions
  - apply *same* hash function  $h1$  to the join attribute of both relations  $R$  and  $S$ . This produces partitions of  $R$  and  $S$  that should be smaller than the available buffer.
  - apply second hash function to a partition of  $R$  and build in-memory hash table
  - load one page of corresponding partition of  $S$
  - scan loaded page and apply  $h2$  to each instance and compare using hash table on partition of  $R$
  - once a page was fully scanned it is discarded (same as partitions of  $R$  after all pages of corresponding partition of  $S$  were compared)



# Hash Join

- ◆ Access paths and indexing
- ◆ Implementation of logical access
- ◆ Query optimization



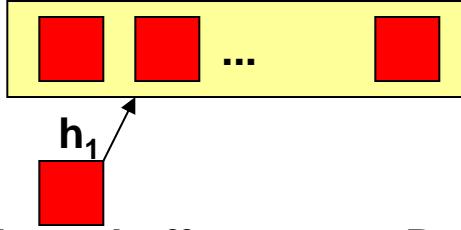
## ■ Cost function:

- Scan  $R$  and  $S$  once, apply  $h1$ , write partitioned  $R$  and  $S$  once:  
$$2b_r + 2b_s$$
- Read each partition of  $R$  once and each page of each  $S$  part.:  $b_r + b_s$
- Apply hash function  $h2$ , perform match, write result set:  $b_{rs}$
- Total cost function:  $3b_r + 3b_s + b_{rs} + (\text{CPU cost for applying } h_1, h_2, \text{ match})$



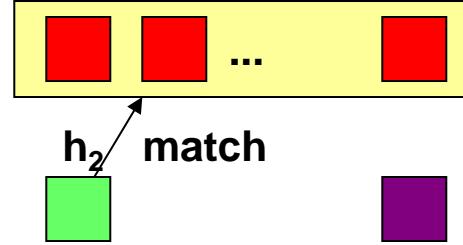
# Memory Requirements

**Hash table for blocks of  $R_i$**



**Input buffer to scan  $R$**

**Hash table for blocks of  $R_i$**



**buffer to scan  $S_i$  Output buffer**

## In partition phase:

- We need 1 output buffer for each partition + 1 input buffer
- Each partition of  $R$  is approx.  $b_r/(B-1)$

## In probing phase:

- Size of in-memory table for probing phase is  $(f^*b_r)/(B-1)$
- $f$  = fudge factor (accounts for increase in memory requirement of hashed vs. non-hashed partition)
- Required buffer  $B > (f^*b_r)/(B-1)+2$
- $\Rightarrow B > \sqrt{f^*b_r}$



# Overflow Handling

- If a partition  $R_i$  doesn't fit into the available buffer a **major performance degradation** will be the result
- To make  $R_i$  fit into buffer, apply hash function recursively (to both  $R_i$  and the corresponding  $S$  partition)
- In join phase join subpartitions pairwise

# Hybrid Hash Join

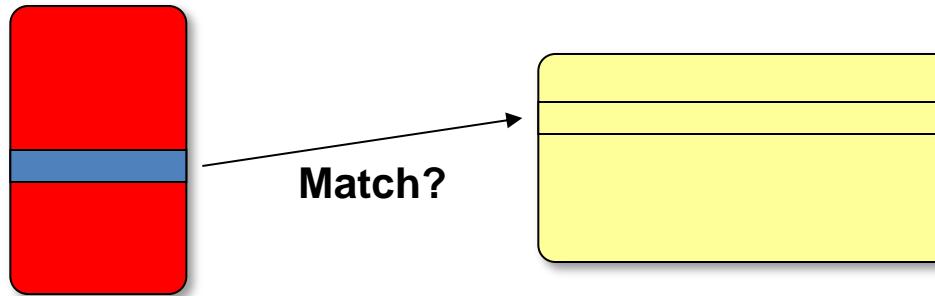
- In **hybrid hash join** we mix the partitioning and the probing phases
- If there is enough extra memory, so that we can build the partitions and hold the first partition in main memory as a hash table while partitioning, we need not write it to disk
- If there is enough space to hash one relation completely, we need only access it **once** for hashing, scan the other relation once (and hash it), match and write results

$$C_{best} = b_r + b_s + b_{rs} + CPU \text{ costs}$$

# Sort-Merge Join

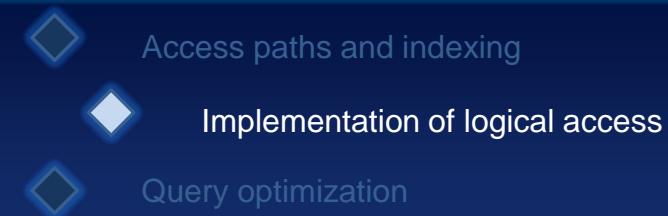
If relations are sorted on the join attribute a very efficient join is possible (especially for key attributes):

- Descend  $R$  and  $S$  and compare tuples
  - since tuples are in sorted order, if no match is found, we can advance and need not consider previous values



- Cost function **sorted**:  $C = b_r + b_s + (b_{rs})$
- Cost function **unsorted**:  $C = b_r + b_s + b_r \log b_r + b_s \log b_s + (b_{rs})$

# Refinements to Sort-Merge Join



- **External sorting:**
  - assume  $m$  buffer pages,  $br$  blocks in relation  $R \rightarrow \lceil b_r/m \rceil$  runs where  $m$  is the number of buffers available for sorting
- Cost for producing sorted relation  $R$ :
  - cost for reading whole relation  $R$ :  $b_r$
  - CPU cost for producing sorted runs:  
$$\lceil (b_r/m) \rceil * b_{fr} * m * \log(b_{fr} * m) = b_{fr} b_r \log(b_{fr} * m)$$
 $b_{fr} * m$  is the number of tuples of type  $r$  in the buffer of size  $m$
  - cost for writing sorted runs:  $b_r$
- Cost for producing sorted relation  $s$  is similar
- Cost for merging sorted runs (if  $m > \lceil b_r/m \rceil + \lceil b_s/m \rceil$ ) taking first block of each sorted run, 1 block for result and making  $n$ -way comparison:  $b_r + b_s + b_{rs}$
- Total cost refined sort-merge join:  $3b_r + 3b_s + b_{rs} + (\text{CPU costs})$

# Hash Join vs. Sort-Merge Join

- There is not ***the best*** join algorithm
- Performance of join algorithms always depends on properties of relations and available buffer
- If  $B > \sqrt{M}$  ( $M$  being the smaller relation)
  - with uniform partitioning hash join costs  $3(M+N)$
- If  $B > \sqrt{N}$  ( $N$  being the larger relation)
  - sort-merge join costs  $3(M+N)$
- If partitions are nonuniform → hash join more expensive
- If available buffers between  $\sqrt{M}$  and  $\sqrt{N}$ , hash join better (only need size of smaller relation)
- If sorted output important (pipelining!) **sort-merge join** better

# General Join Conditions

- So far have analyzed only **simple equality conditions**
- Often need to compare on multiple attributes or use inequality comparison
- For multiattribute conditions ( $R.x=S.x \wedge R.y=S.y$ )
  - **sort** (sort-merge-join) or **hash** (hash-join) on combination of attributes
  - build index on combined attributes (index join)
- For **inequality condition**
  - sort-merge and hash joins not applicable
  - index join needs a B+ tree
  - nested loop not affected

# Aggregate Operations

- **Basic SQL aggregate operators:** AVE, MAX, MIN, SUM, COUNT
  - scan relation once and keep running information
- **Aggregates in combination with GROUP BY**
  - must compute aggregate for each partition generated by GROUP BY
- **Aggregates with GROUP BY using sorting:**
  - sort relation, scan looking for group boundaries (can be optimized by aggregating while sorting):
  - $C_{best} = br + \text{CPU costs for sort}$      $C_{worst} = 3 br + \text{CPU costs for sort}$
- **Aggregates with GROUP BY using hashing:**
  - produce hash table with entries <grouping value, running info>
  - most likely will fit in main memory (small entries, only one per grouping value), otherwise hash twice :  $C_{best} = br + \text{CPU costs for hash}$



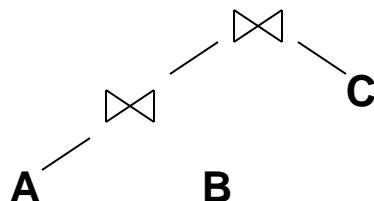
# The Impact of Buffering

- **Buffer size is critical** for the performance of relational ops.
- Concurrently executing operations compete for the buffer
- **If indexes are used:**
  - finding a page that is requested repeatedly in the buffer pool depends on buffer pool and replacement strategy
  - if an unclustered index is used, likelihood that each tuple accessed requires a new page in the buffer is high (fills quickly)
- **Patterns** can be exploited by using adequate replacement policy or reserving enough buffer
  - LRU is worst strategy for nested loop join (use MRU)
  - index join: sort on non-indexed relation maximizes reutilization of access paths



# Pipelined Evaluation

- In **pipelined evaluation** the intermediate results are not written (materialized) to a temporary relation
- Intermediate results are passed ***on the fly*** to the next operator, e.g.:
  - **composite selection predicate** for which an index exists only on one part of the predicate - could be seen as 2 sequential selections
  - **projection** in which key attribute is part of result (no duplicate elim.)
  - **pipelined joins**: C acts as innermost relation (in nested loop join); B acts as inner relation at next level; A is scanned and results of that join (tuples) are pipelined to the join with C



# The Iterator Interface

- To pass tuples from one operator to the next, an **iterator interface** is required
- The iterator interface acts *similarly* to a cursor
- Iterator interface has operations *open*, *get\_next*, *close*
  - *open* allocates buffers and passes parameters (e.g. selection condition)
  - *get\_next* calls *get\_next* function on each input node, calls operator-specific code to process the input tuples, keeps track of progress
  - *close* deallocates buffers and state information
- The iterator interface hides implementation details and will pipeline without intermediate materialization whenever possible

# Query Optimization

- Basic introduction well explained in Ramakrishnan/Gehrke

Ramakrishnan, R., Gehrke, J.

*Database Management Systems*

2<sup>nd</sup> Ed., McGraw Hill, 2000

3<sup>rd</sup> Ed., McGraw Hill, 2003 (updated 2007)

- Older but still classic overview paper by Goetz Graefe in ACM Computing Surveys

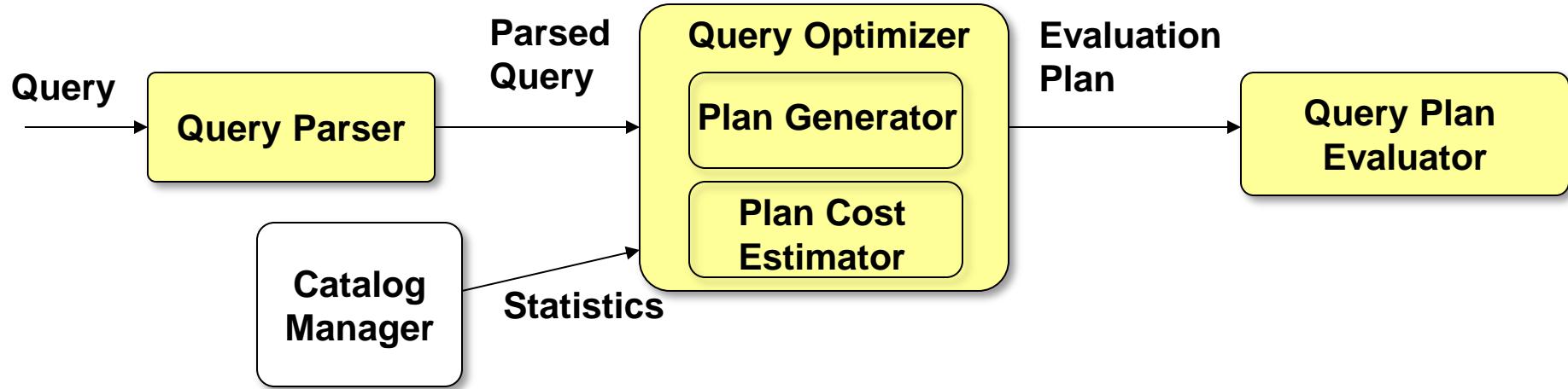
Graefe, G.

*Query evaluation techniques for large databases*, ACM Comp. Surv., 25:2, 1993, pp. 73-170



# Relational Query Optimization

- Implementation of logical access
- Query optimization
- Transaction Management



- **Main functions of query optimizer:**
  - enumerate alternative plans for evaluating expression (subset of all possible plans, exhaustive enumeration not economical)
  - estimate the cost of each plan, using statistics from catalog
  - choose plan with least estimated cost



# Building a Query Tree

- **Scanner** - identifies individual language elements (tokens)
- **Parser** - checks syntactic correctness of query
  - constructs internal query tree, translates external names via catalog
  - checks syntactic correctness, availability of relations and attributes
  - checks access rights
    - via capability matrix for content-independent access control
    - via query modification for content-dependent access control
  - checks integrity constraints
    - syntactic constraints tested immediately (format, type)
    - for constraints that must be checked at run-time, query is modified or triggers are set
- Result is a modified (extended) **query tree** - leaves contain operands (relations or constants), internal nodes operators

# Steps and Criteria for Query Optimization



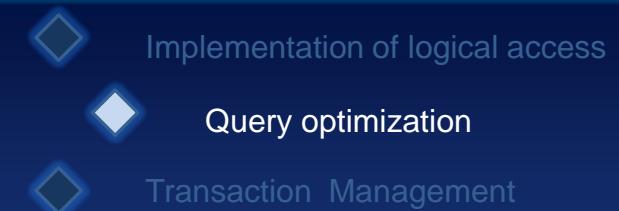
- Optimizer establishes set of possible query plans
- **Cost criteria:**
  - I/O operations (I/O-bound processes, simple optimizers)
  - CPU-time (CPU-bound processes, more important than assumed in text books!)
  - weighted combination of I/O and CPU-time (used in good industrial optimizers)
- **Elements of optimization:**
  - sequence of operations (e.g. select before join, high selectivity first)
  - available access paths (e.g. hash tables, B+ trees)
  - alternative implementation of operators (e.g. nested-loop vs. hash-join)



# Code Generation

- **Code generator** generates program code for query
  - program-code stored as module in module library
  - parameters for individual instantiations are passed
  - code for formatting of results is inserted
- Alternatively **queries** can be interpreted
  - late binding, therefore high data independence
  - expensive - 10 to 20 times more expensive than compiled code, repeated access to catalog
- Compilation with storage as modules preferable
  - lazy evaluation when access paths or DB structure change
  - transparent to user, user only notices delay in first execution after change

# A running example for Query Optimization



Example for optimization taken from *Ramakrishnan*:

**Sailors**(sid:*integer*,sname:*string*,rating:*integer*,age:*real*)

**Boats**(bid:*integer*,bname:*string*,color:*string*)

**Reserves**(sid:*integer*,bid:*integer*,day:*date*,rname:*string*)

Each tuple of Reserves = 40 Bytes

Each page holds 100 Reserves tuples, 1000 pages of Reserves

Each tuple of Sailors = 50 Bytes

Each page holds 80 Sailors tuples, 500 pages of Sailors



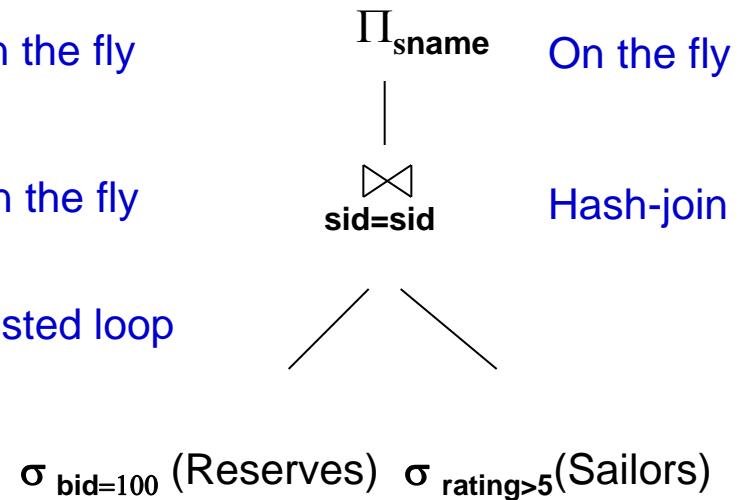
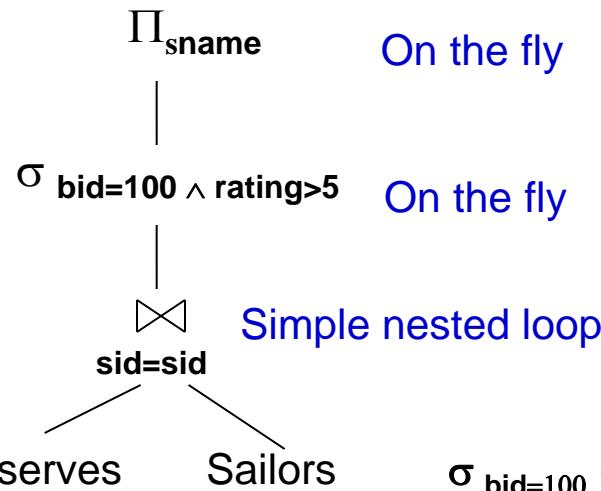
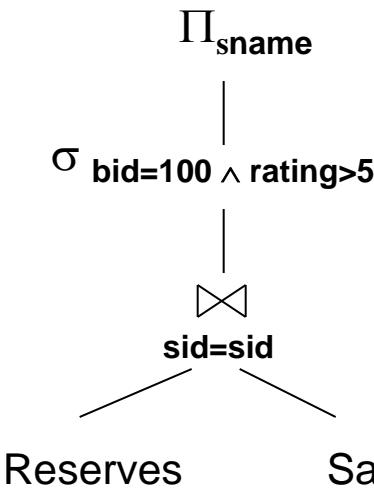
# Query Evaluation Plans

Query evaluation plan consists of extended relational algebra tree, annotations that indicate the access methods to be used and implementation method for the relational operators

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

$$\Pi_{\text{sname}}(\sigma_{\text{bid}=100 \wedge \text{rating}>5}(\text{Reserves} \bowtie_{\text{sid}=\text{sid}} \text{Sailors}))$$


# Query Evaluation Plans (cont.)

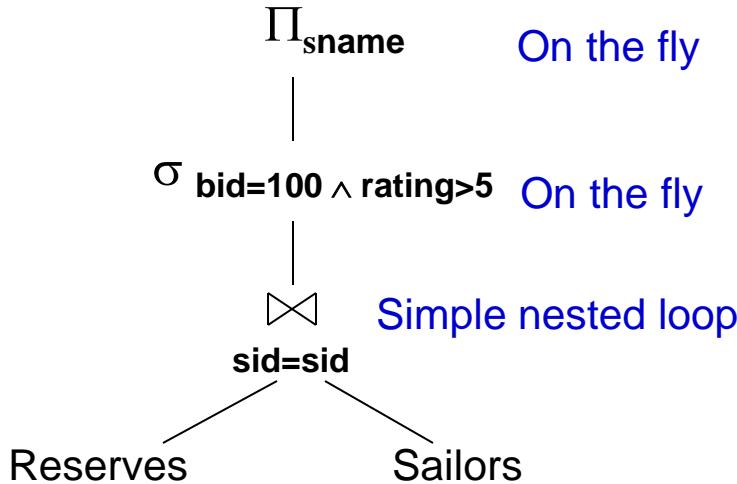
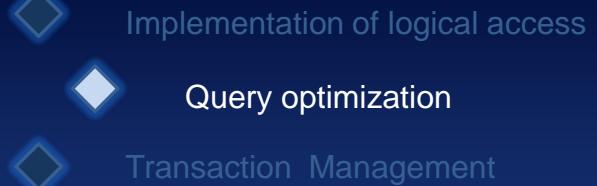
$$\Pi_{\text{sname}}(\sigma_{\text{bid}=100 \wedge \text{rating}>5}(\text{Reserves} \bowtie_{\text{sid}=\text{sid}} \text{Sailors}))$$


**Query as Relational Algebra Tree**

**Query Evaluation Plan for Sample Query**

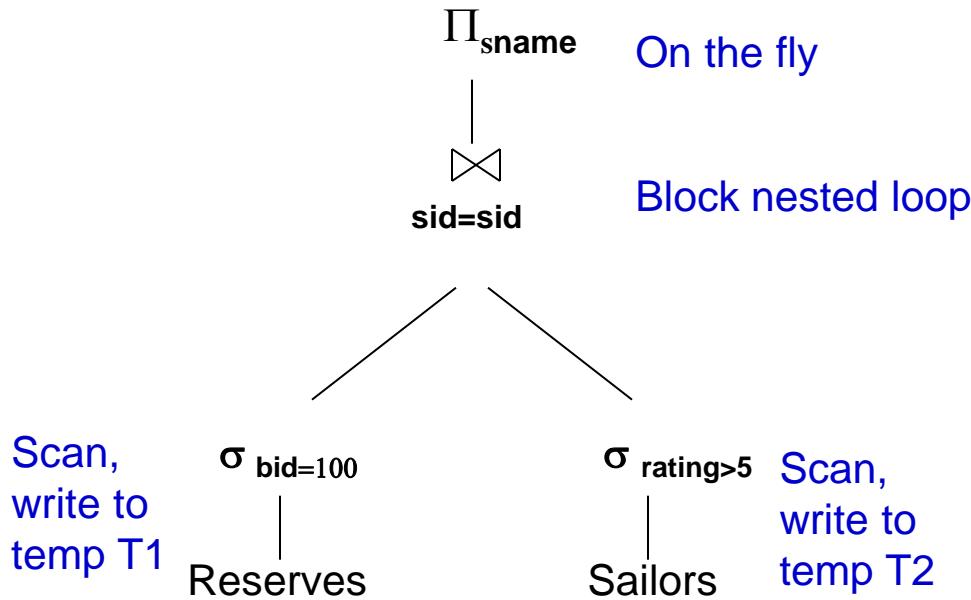
**Optimized Query Evaluation Plan for Sample Query**

# Alternative Plans - A Motivating Example



- **Cost for this plan is:**
  - join:  $1000 + 1000 * 500 = 501\ 000$  page I/Os
  - selection and projection are done on the fly, i.e. no writing of intermediate results, therefore no additional I/O costs
- An **even more naive plan** would materialize intermediate results and/or use cartesian product instead of join

# A Better Plan

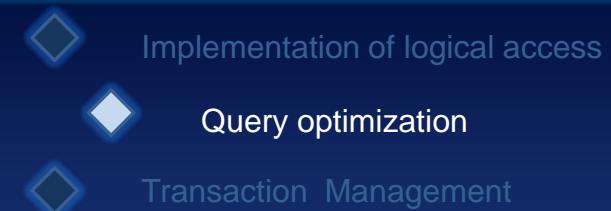


- Apply selection early to reduce input to join
- Scan of reserves: 1000 I/Os
- Assuming selectivity factor of 0.01 (there are 100 boats), T1 requires 10 I/Os
- Scan of Sailors: 500 I/Os
- Uniform distribution across 10 ratings, T2 requires 250 I/Os
- 5 buffer pages: 1 scan of T1 (10 I/Os) and  $\lceil 10/3 \rceil = 4$  scans of T2 = 1000 I/Os
- $1000 + 500 + 10 + 250 + 10 + 1000 = 2770$  I/Os

# Other Optimizations

- **Cost of join** can be reduced by pushing Projection past join:
  - only sid in  $T1$  and sid and sname in  $T2$  are needed
  - by projecting sid in  $T1$ ,  $T1$  fits in 3 pages, therefore block nested loop with 5 buffers can be done with single scan of (a smaller)  $T2$
  - cost of join drops to 250 I/Os and total cost to approx. 2000 I/Os
- If indexes are available even better plans are possible

# Heuristic Query Optimization



- **Heuristic query optimization** is used to enumerate alternative query trees
- Heuristic optimization applies *transformation rules* to produce *equivalent* query trees
- The main goal of heuristic query optimization is to **minimize intermediate results**



# Transformation Rules

- Transformation rules of relational algebra must transform a query tree into an equivalent query tree
- There exist proven transformations that guarantee equivalence
- **R1: Cascading of selections:**  
conjunctive selection predicates can be transformed into a sequence of individual selections
$$\sigma_{c_1 \wedge c_2 \wedge \dots \wedge c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_n}(R))\dots))$$
- **R2: Commutativity of selection**
$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

# Transformation Rules (cont.)

## ■ R3: Cascading of Projection

In a cascade of projections only the last is relevant

$$\pi_{\text{list1}}(\pi_{\text{list2}}(\dots(\pi_{\text{listn}}(R))\dots)) \equiv \pi_{\text{list1}}(R)$$

## ■ R4: Commutativity of Selection and Projection

If selection predicate only contains the attributes that are part of the projection list, selection and projection commute

$$\pi_{A1, A2, \dots, An}(\sigma_c(R)) \equiv \sigma_c(\pi_{A1, A2, \dots, An}(R))$$

## ■ R5: Commutativity of Join and Cross-product

Assuming named attributes, position in resulting relation is immaterial

$$R \bowtie_c S \equiv S \bowtie_c R$$



# Transformation Rules (cont.)

## ■ R6: Commutativity of Selection and Join

For the case where selection predicate c only involves the attributes of a single relation (e.g. R)

$$\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S$$

For the case where selection predicate c involves attributes of R and S but there is a valid conjunctive form ( $c_1 \wedge c_2$ )

$$\sigma_c(R \bowtie S) \equiv (\sigma_{c_1}(R)) \bowtie (\sigma_{c_2}(S))$$

*Commutativity also applies to Cartesian product and selection.*

# Transformation Rules (cont.)

- **R7: Commutativity of Projection and Join**

If projection list A1, A2, ..., Am from R and projection list B1,B2,...,Bn from S contain all the attributes needed for the join condition

$$\pi_L(R \bowtie_c S) \equiv \pi_{A1, A2, \dots, Am}(R) \bowtie_c \pi_{B1, B2, \dots, Bn}(S)$$

If join condition contains additional attributes, these attributes must be included in the projection list and at the end one additional projection must be carried out to eliminate them

- **R8: Commutativity of Set Operations:**

Union and Intersection are commutative, Set difference is not commutative

# Transformation Rules (cont.)

- **R9: Associativity of Join, Carthesian Product, Union and Intersection**

Each of these operations ( $\theta$ ) is individually associative

$$(R \theta S) \theta T \equiv R \theta (S \theta T)$$

- **R10: Commutativity of Selection with Set Operations**

Selection commutes with Intersection, Union and Set Difference

$$\sigma_c(R \theta S) \equiv (\sigma_c(R)) \theta (\sigma_c(S))$$

# Transformation Rules (cont.)

- **R11: Commutativity of Projection and Set Operations**

Projection commutes with Union

$$\pi_L(R \theta S) \equiv (\pi_L(R)) \theta (\pi_L(S))$$

What about Intersection or Set Difference?

- Transformation of predicates according to **DeMorgan's laws**

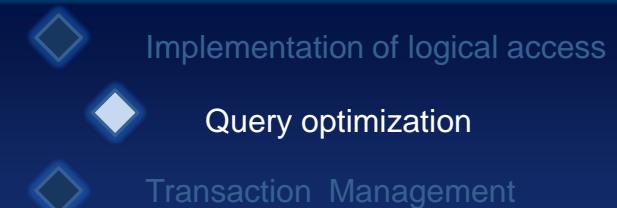
$$c = \text{NOT}(c_1 \text{ AND } c_2) \equiv (\text{NOT}c_1) \text{ OR } (\text{NOT}c_2)$$

$$c = \text{NOT}(c_1 \text{ OR } c_2) \equiv (\text{NOT } c_1) \text{ AND } (\text{NOT } c_2)$$



# Steps in Heuristic Optimization

## - Precedence



- **The precedence for application of rules is**
  - *break up* conjunctive selection predicates into a cascade of individual predicates for flexibility (R1)
  - *apply* commutativity rules to move selections as close as possible to the leaves (R2,R4,R6,R10)
  - *use* associativity to move binary operations that produce small result sets closer to the leaves (R9)
  - *combine* carthesian product followed by a selection into a join
  - *reduce* size of intermediate relations through projection, reduce data transfer by executing  $\pi$  as far down in query tree as practical
  - *group* operations to execute more than one operation with one I/O
- **Rule of thumb:** Selection and Projection close to leaves, most restrictive joins first



# Estimating the Cost of a Plan

- **Two basic tasks:**
  - for each node in query tree, estimate the cost of executing the operation
  - for each result estimate the size of the result set and determine if it is sorted
- **Estimates are rough approximations**
  - optimizer will not find the best plan, but a good plan and will avoid the worst plans
  - the better the statistics, the better the cost estimate, but statistics are costly to maintain

# Statistics

- **Statistics are kept in system catalog and updated periodically**
  - **Cardinality:** the number of tuples  $NTuples(R)$  for each relation  $R$
  - **Size:** the number of pages  $Npages(R)$  for each relation  $R$
  - **Index Cardinality:** number of distinct key values  $Nkeys(I)$  for each index  $I$
  - **Index Size:** the number of pages  $INPages(I)$  for an index  $I$  (and/or leaf pages in B+ tree)
  - **Index Height:** the number of non-leaf levels  $Iheight(I)$  in a tree index
  - **Index Range:** minimum  $Ilow(I)$  and maximum  $Ihigh(I)$  key value for each index
  - MaxVal, MinVal and AttribCard



# Estimation of Result Sizes

Consider following query block:

```
SELECT attribute list  
FROM relation list  
WHERE term1 ∧ term2 ∧ ... ∧ termn
```

- Maximum size is given by product of cardinality of relations in `FROM` clause
- Selection criteria restrict the number of tuples in result set
- Estimate = maximum size \* reduction factors
- Must estimate reduction factors

# Reduction Factors

- **Column = value**       $1/N_{keys}(I)$  if index exists  
 $1/(MaxVal - MinVal)$  or  $1/AttribCard$
- **Column1 = Column2**       $1/\text{MAX}(N_{keys}(I_1), N_{keys}(I_2))$   
if no index exists either use  
AttribCard or fudge factor (0.1)
- **Column > Value**       $(High(I)-value)/(High(I)-Low(I))$   
 $(MaxVal-value)/(MaxVal - MinVal)$
- **Column IN <valuelist>** approximate by using  
 $\text{column}=\text{value} * \# \text{ values in list}$
- **Projection**      ratio of tuple sizes after and before

# Histograms

- Previous estimates assumed uniform distribution
- For attributes with nonuniform distribution this leads to poor estimates

→ Introduce histograms

Histograms divide the range of the attribute into buckets and maintain statistics for each bucket

- Equiwidth histograms maintain buckets of equal size (width), i.e. the range is divided into buckets and a uniform distribution is assumed within bucket
- Equidepth histograms define buckets of equal size (number of tuples) by grouping a variable number of adjacent values

Most commercial optimizers use equidepth histograms

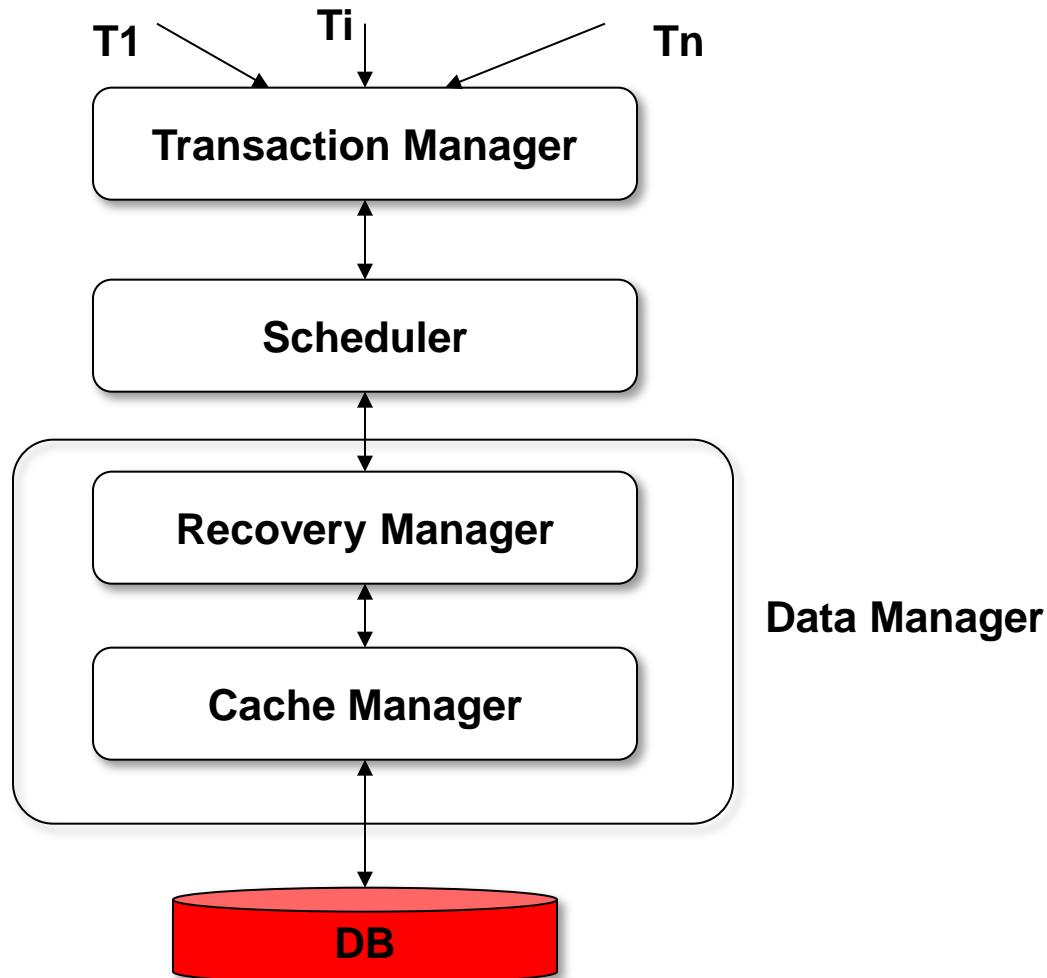


# Calculating the Cost

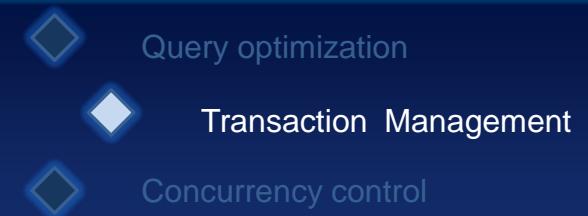
- Make **enumeration of alternatives**
  - from tree obtained from *heuristic optimization*, generate alternatives based on
    - available access paths
    - available implementation of operators
    - estimate of result size
    - ordering and availability of iterators
    - policies
- Apply cost functions for each implementation with specific data for relations involved

# Transaction Management

- Query optimization
- Transaction Management
- Concurrency control



# Scheduler



- **Scheduler is responsible** for the concurrent execution of transactions
- **Scheduler determines** the proper order of execution of the individual commands coming from upper layers
- **Scheduler has 3 options:**
  - *execute command*: command is passed on to the Data Manager; status code or value read are returned
  - *reject command*: causes transaction to abort
  - *delay execution of command*: command is placed in waiting queue from where scheduler retrieves it for execution later



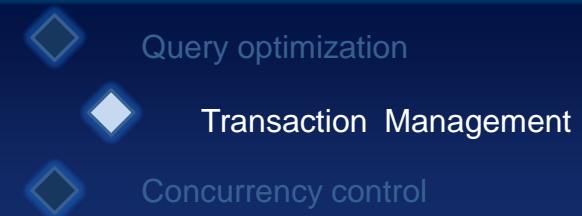
# Scheduler: Ordering of Executions



- **2 possible mechanisms** to determine order of execution:
  - **FIFO queues**: cause unnecessary delays through strict sequential execution. With more than 2 executing modules (e.g. in distributed environment) no unique sequence is possible
  - **Handshake**: Upper layer waits for confirmation that command was executed:  $\langle p - \text{ack}(p) - q \rangle$   
Parallelizable commands do not require ack.
- Will assume handshake as mechanism of choice
- Permissible command sequences depend on transaction model and correctness criterion used



# Transactions



- Transactions are minimal processing units in the DBS and are bracketted by
  - **BOT** - begin of transaction
  - **EOT** - end of transaction
- Between **BOT** and **EOT** markers a transaction consists of an arbitrary sequence of semantically correct **DML** and **PL** statements

**Def:** A **transaction** is an atomic process that transforms a database from one consistent state into another consistent state



# Consistency



- **Database consistency:** a database is consistent if it complies with all the consistency constraints defined on it.
- **Transaction consistency:** a transaction maps a consistent database state to another consistent database state; concurrent transactions should not interfere and produce inconsistent states
- Database must be consistent before and after the execution of a transaction



# ACID Properties



- Transactions are expected to comply with the **ACID properties**:
  - **Atomicity** - transaction executes all or nothing
  - **Consistency** - transactions do not produce inconsistent states
  - **Isolation** - changes made by a transaction become visible only after the commit
  - **Durability** - changes are permanent

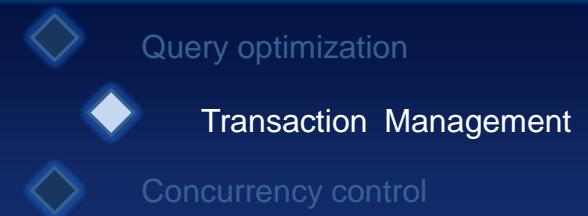


# Transactions: Termination

- *Transactions always terminate:*
  - **normal termination** - commit: changes are permanent in the DB
  - **abnormal termination** - abort: changes are rolled back, DB remains in the same state as before the BOT
- *Abort can be initiated by*
  - user or application program
  - system
- Transactions that were aborted can be
  - **restarted** only if transaction was aborted due to HW or system failure (e.g. deadlock)
  - **removed** faulty transactions are removed (e.g. arithmetic error)



# Transaction Models



- **Transaction model** determines behavior and interface to the transaction manager
  - object structure
  - transaction structure
- **Object structure:**
  - flat objects (e.g. tuples in a relation)
  - complex objects (e.g. aggregates)
    - complex objects may cause indirectly a nesting of transactions
  - passive objects (all objects in standard DBs)
  - active objects (objects with triggerable ECA rules)
    - active objects cause transaction nesting



# Transaction Structure



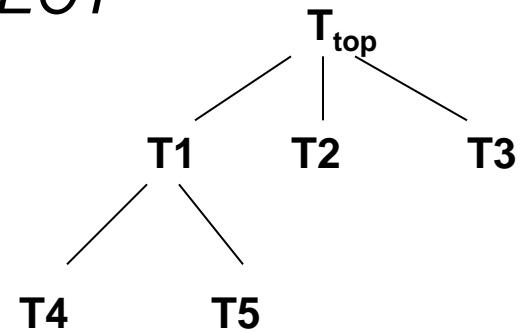
- **Flat transactions** - most commercial systems, single thread of control

$T1: \text{BOT} \rightarrow DML1 \dots DMLn \rightarrow EOT$

- **Nested transactions**

- Transaction has tree structure
- commit dependencies
- abort dependencies
- visibility rules

- **CD, AD, VR** determine if transaction is
  - closed nested
  - open nested



# Chained Transactions (Transaction chopping → later)

- Problem: If the system crashes during the execution of a long-running transaction, considerable work can be lost
- Chained Tx: new Tx starts after the previous commits or aborts
  - Chaining allows a transaction to be decomposed into subTx with intermediate commit points
  - Database updates are made durable at intermediate points → less work is lost in a crash
  - Savepoint: explicit rollback to arbitrary savepoint; all updates lost in a crash
  - Chaining: abort rolls back to last commit; only the updates of the most recent transaction lost in a crash

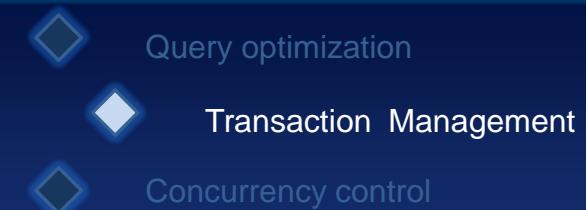
S1  
S2  
S3  
**commit**



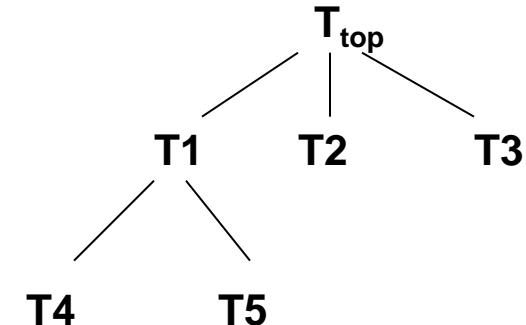
**S1;** → *update recs 1 - 1000*  
**commit;**  
**S2;** → *update recs 1001 - 2000*  
**commit;**  
**S3;** → *update recs 2001 – 3000*  
**commit;**



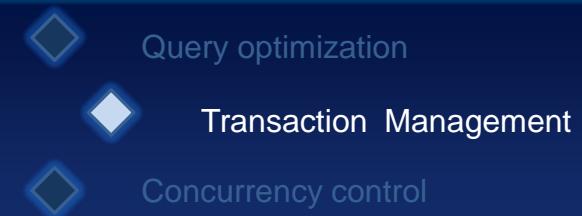
# Closed Nested Transaction Model



- **First proposed by Moss**
  - execution may only occur in leaves
  - commit is always through the parent
  - if parent aborts, children abort
  - if child aborts, parent may do error handling
  - results are only visible after they are committed (e.g. T5 sees changes done by T4 after T4 commits to T1, all changes become visible to the outside when T<sub>top</sub> commits)
  - closed nested transactions can be rolled back
- Models proposed by *Rothermel* and *Härder* and by *Dayal* and *Hsu* are somewhat more flexible



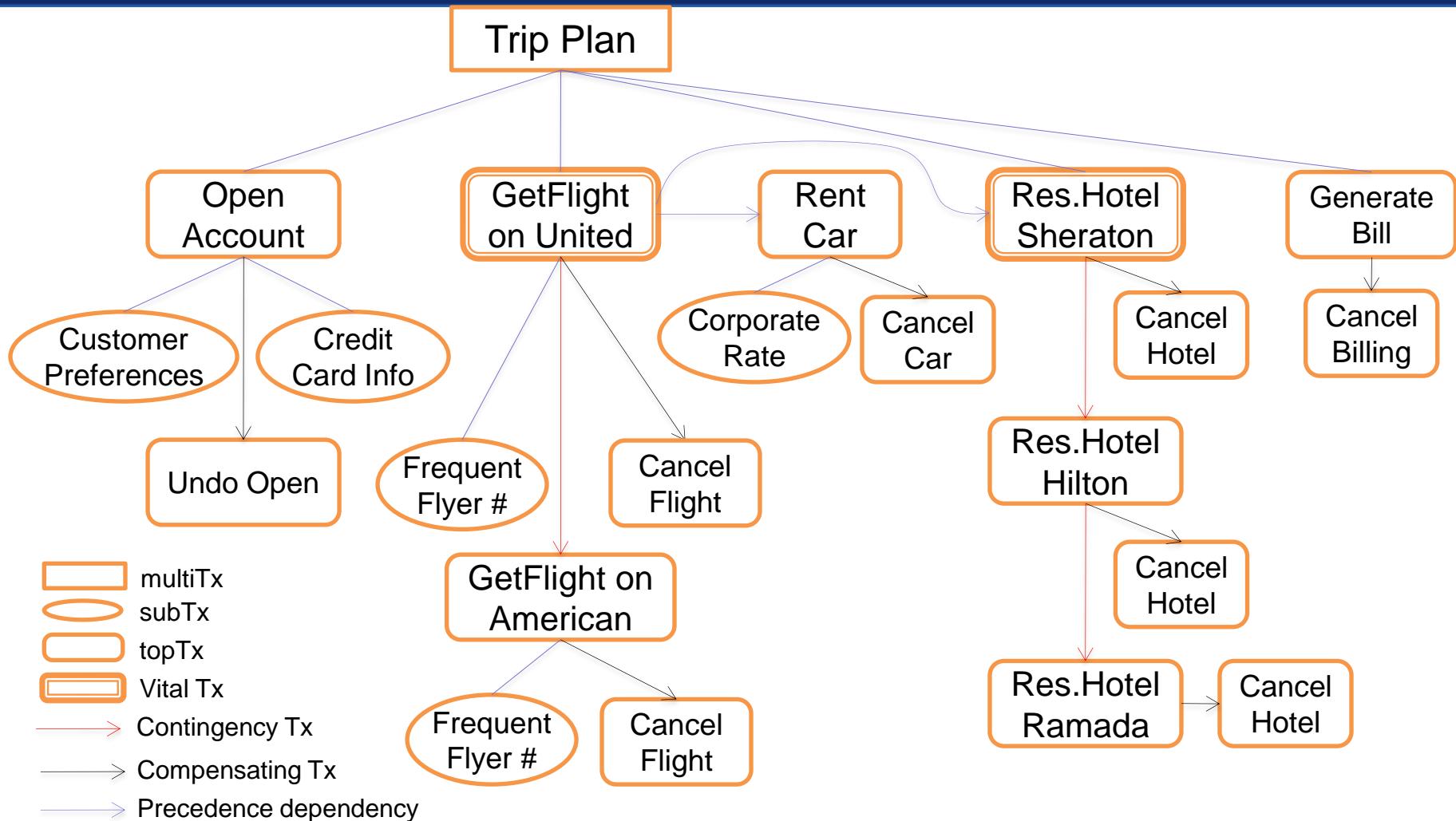
# Open Nested Transactions



- **Open nested transactions** also have tree structure
- Visibility rules (isolation) are relaxed: results of subtransactions may be visible to the outside
- Because subtransactions may commit independently of the parent transaction, rollback is no longer possible, instead compensating transactions are used
  - compensating transaction is semantically inverse transaction
  - compensating transactions may not always exist (irreversible side effects)
- Subtransactions may be vital or not (abort dependency of parent transaction on vital subtransaction)

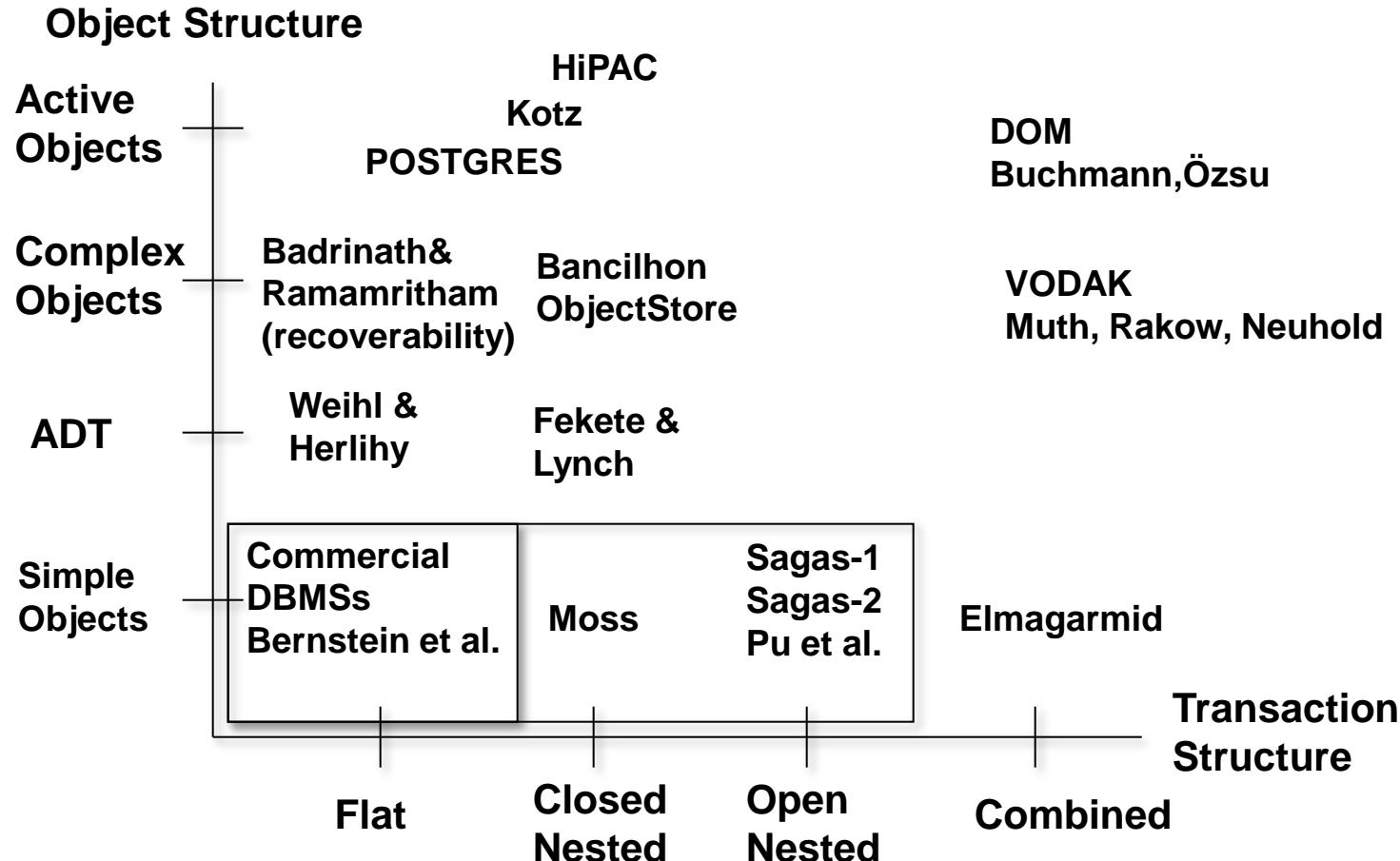


# Example of Open Nested Transaction (DOM Model)



# Space of Transaction Models

- Query optimization
- Transaction Management
- Concurrency control



# Readings on Transaction Models



- A. P. Buchmann, M. T. Ozu, M. Hornick, D. Georgakopoulos, F. A. Manola, **The DOM Transaction Model**, in A. Elmagarmid (Ed.), **Database Transaction Models for Advanced Applications**, Morgan-Kaufmann, 1992
- P. Chrysanthis, K. Ramamritham, **ACTA: a framework for specifying and reasoning about transaction structure and behavior**, SIGMOD 1990, Atlantic City, NJ



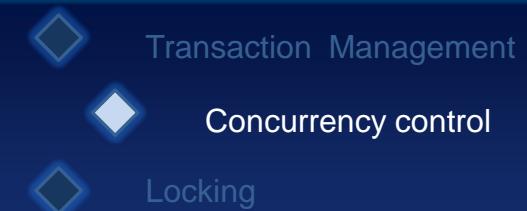
# Correctness Criteria

- Correctness Criteria determine legal command sequences
- Sequential execution of transactions is correct by definition (since each transaction is individually correct)
  - every transaction execution that is **equivalent** to a sequential execution is correct = ***serializability***
- Serializability is one of many possible correctness criteria (but the best understood)
- Some applications (CAD,WF) need less stringent correctness
  - *Transaction classes* (Garcia Molina 83)
  - *Breakpoints* (Lynch 83, Farrag & Özsu 89)
  - *Quasiverbalizability* (Du & Elmagarmid 89)
  - *Multidatabase serializability* (Barker & Özsu 89)



# Serializability Theory

## (Bernstein, Hadzilacos, Goodman)



### ■ Transaction

- Sequence of read and write operations  $r_i[x]$  and  $w_i[x]$  terminated by a commit  $c_i$  or an abort  $a_i$ ;

$$r_i[x] \rightarrow w_i[x] \rightarrow c_i$$

- $T_i: (\Sigma_i, <_i)$  where  $\Sigma_i$  represents the set of operations in transaction  $T_i$  and  $<_i$  is the order relation  
For simplicity  $T_i$  is used instead of  $(\Sigma_i, <_i)$

$r_2[x]$

$r_2[y]$

$w_2[z] \rightarrow c_2$

$T_2: \{r_2[x], r_2[y], w_2[z], c_2\}$

$<_2 = \{(r_2[x], w_2[z]), (r_2[y], w_2[z]), (r_2[x], c_2), (r_2[y], c_2), (w_2[z], c_2)\}$

Operations that can be executed in parallel can be represented as a partial order

# Transaction Definition

**Def:** A transaction  $T_i$  is a partial order with order relation  $<_i$ , such that

- $T_i \subset \{r_i[x], w_i[x] \mid x \text{ is a data element}\} \cup \{a_i, c_i\}$
- $a_i \in T_i$  iff  $c_i \notin T_i$
- for each operation  $p$  in  $T_i$ ,  $p <_i c_i$  or  $p <_i a_i$
- for  $r_i[x], w_i[x]$  in  $T_i$  there is a unique order  $r_i[x] <_i w_i[x]$  or  $w_i[x] <_i r_i[x]$

- Transactions are abstracted into read and write operations
- Transactions described independently of content of data elements → transaction analysis must be independent of content and may not assume any initial values
- Transaction abstraction contains enough syntactic info. for the scheduler to produce serializable schedules

# Histories

- **Histories** describe concurrent execution of a set of transactions
- **Histories  $\neq$  log**
- Histories contain all operations (incl. reads), log not
- History is a partial order that describes
  - the relative execution order of all operations in a transaction
  - the order of all conflicting operations
- 2 operations are *incompatible* when both refer to the same data element and at least one is a write operation
  - $r[x]$  conflicts with  $w[x]$
  - $w[x]$  conflicts with  $w[x]$  and  $r[x]$
- **Order among conflicting operations is important!**



# Complete Histories

**Def:** The complete history of a set of transactions

$T = \{T_1, T_2, \dots, T_n\}$  is a partial order with order relation  $<_H$  such that

- 1)  $H = U_n^{i=1} T_i$  (all operations in T)
- 2)  $<_H \supseteq U_n^{i=1} <_i$  (op. sequence in  $T_i$  preserved)
- 3) for conflicting operations  $p$  and  $q$ ,  $p <_H q$  or  $q <_H p$

- Histories may be incomplete (e.g. in case of system failure)
- A history is a prefix of a complete history

# Representation of Histories

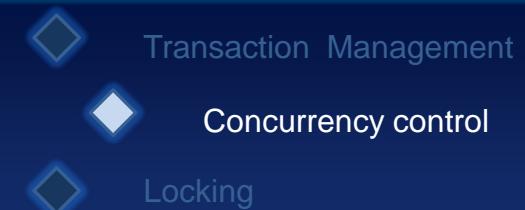
- Histories (like transactions) can be represented as directed graphs

$$T_1 = r_1 [x] \rightarrow w_1 [x] \rightarrow c_1$$
$$T_3 = r_3 [x] \rightarrow w_3 [y] \rightarrow w_3 [x] \rightarrow c_3$$
$$T_4 = r_4 [y] \rightarrow w_4 [x] \rightarrow w_4 [y] \rightarrow w_4 [z] \rightarrow c_4$$

- A complete history for  $T = \{T_1, T_3, T_4\}$  is

$$\begin{matrix} r_3 [x] & \rightarrow & w_3 [y] & \rightarrow & w_3 [x] & \rightarrow & c_3 \\ \uparrow & & \uparrow & & & & \end{matrix}$$
$$\begin{matrix} H_1 = r_4 [y] & \rightarrow & w_4 [x] & \rightarrow & w_4 [y] & \rightarrow & w_4 [z] & \rightarrow & c_4 \\ \uparrow & & & & & & & & \end{matrix}$$
$$r_1 [x] \rightarrow w_1 [x] \rightarrow c_1$$

# Representation of Histories (cont.)



- An **incomplete** history for the same set of transactions

$$r_3 [x] \rightarrow w_3 [y]$$

↑              ↑

$$H'_1 = r_4 [y] \rightarrow w_4 [x] \rightarrow w_4 [y]$$

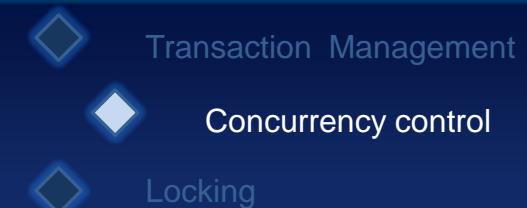
↑

$$r_1 [x] \rightarrow w_1 [x] \rightarrow c_1$$

- A transaction is **committed (aborted)** in  $H$  if  $c_i (a_i)$  is in  $H$
- A transaction for which neither  $c_i$  nor  $a_i$  is in  $H$  is **still active**
- Complete histories don't contain active transactions
- A committed projection  $C(H)$  is the history over  $T$  that only contains the operations of normally completed transactions



# Serial and Serializable Histories



- A history  $H_s$  is **serial**, if for each pair of transactions  $T_i$  and  $T_k$  all the operations of  $T_i$  are ordered before those of  $T_k$  or viceversa
- First attempt of definition for serializability:  
A history  $H$  is serializable (SR) if it is equivalent to a serial history  $H_s$
- **Problem with definition:**
  - $H$  is a prefix history while  $H_s$  must be complete
  - An incomplete history may contain operations of incomplete transactions that may lead to inconsistent DB states
- A history  $H$  is **serializable** (SR) if the committed projection  $C(H)$  is equivalent to a serial history  $H_s$ .



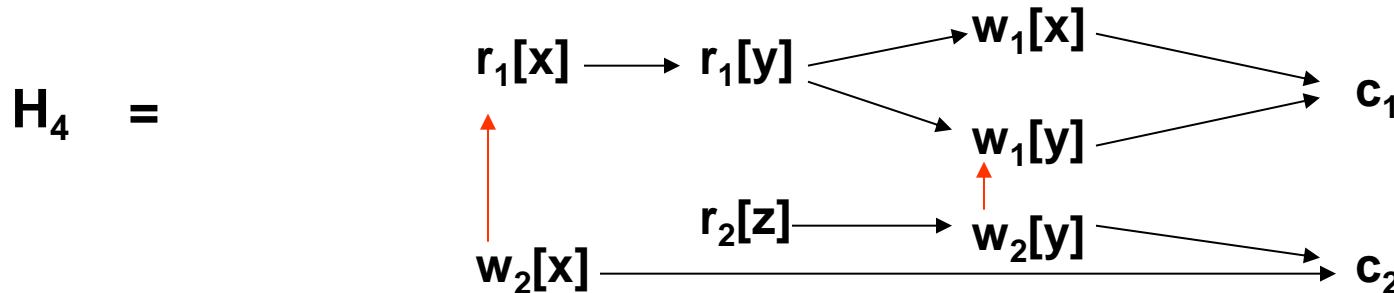
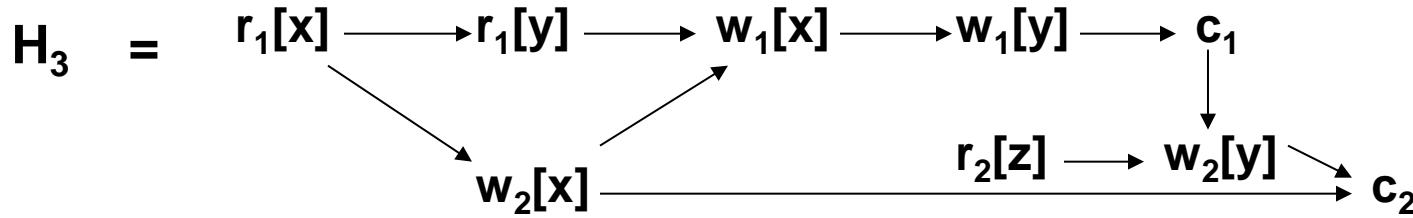
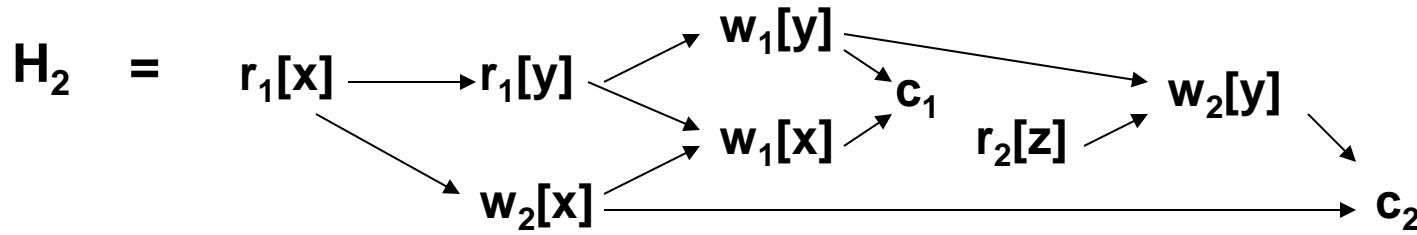
# Serializable Histories

- Definition is based on notion of equivalence with a serial history
  - conditions for equivalence
- **Two histories  $H$  and  $H'$  are (conflict) equivalent if**
  - they are defined over the same set of transactions and have the same operations
  - the *conflicting* operations of the non-aborted transactions are ordered in the same order, i.e. for  $p_i$  in  $T_i$  and  $q_k$  in  $T_k$  and  $(a_i, a_k) \notin H$ , if  $p_i <_H q_k$  then  $p_i <_{H'} q_k$

**Note:** only the order of the conflicting operations is important for serializability

# Examples of Equivalent and Non-equivalent Histories

- Transaction Management
- Concurrency control
- Locking

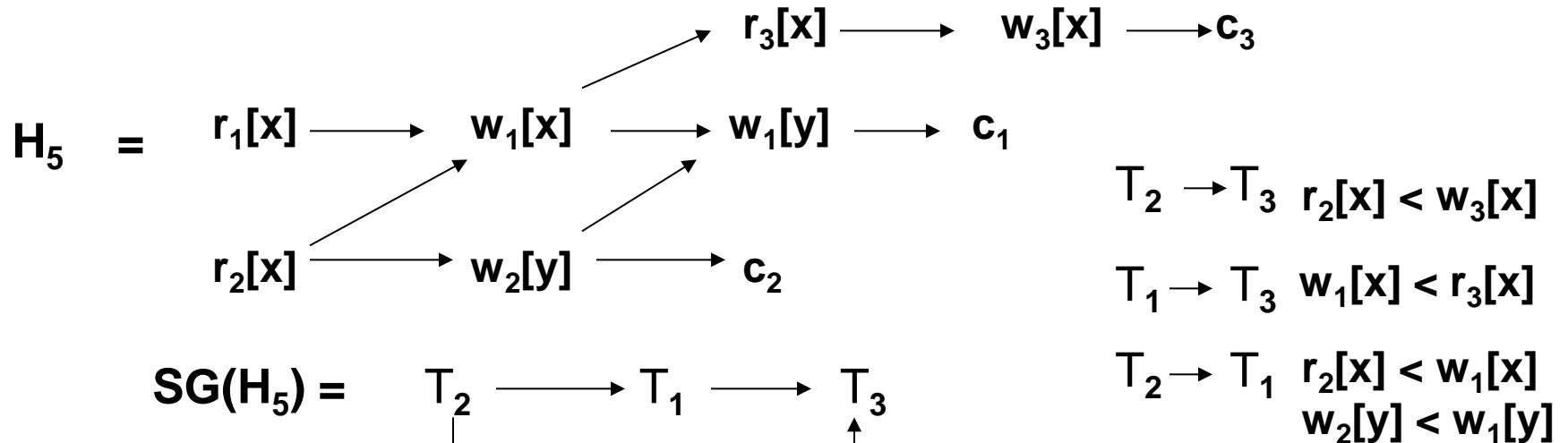


$H_2 \equiv H_3$

$H_4$  not equivalent

# Serialization Graphs

- The execution of a set of transactions can be tested by analyzing the **serialization graph  $SG(H)$**
- $SG(H)$  can be derived from the history.
- Its nodes consist of completed transactions in  $H$
- The edges of the graph represent conflicting operations



# Serialization Graphs

- Since edges in SG represent the order of conflicting operations, the same order must be preserved in a serial execution
- Topological sort will produce all the serial executions that preserve order of conflicting operations
- **Topological sort:**
  - identify vertex with in-degree 0 (no inward edge)
  - eliminate that vertex from graph with all its outward edges
  - insert that vertex as next vertex in the topological sort
  - repeat until all vertices sorted (there is a topological sort) or no vertex with in-degree 0 exists (there is a cycle, no t.s. possible)

# Topological Sort

Topological Sort 1

$V_1$

$V_1 V_2$

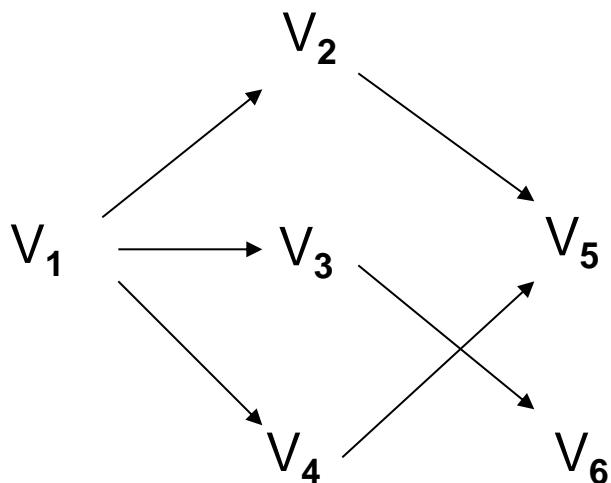
$V_1 V_2 V_3$

$V_1 V_2 V_3 V_4$

$V_1 V_2 V_3 V_4 V_5$

$V_1 V_2 V_3 V_4 V_5 V_6$

Graph



Topological Sort 2

$V_1$

$V_1 V_3$

$V_1 V_3 V_2$

$V_1 V_3 V_2 V_4$

$V_1 V_3 V_2 V_4 V_6$

$V_1 V_3 V_2 V_4 V_6 V_5$

# Serializability Theorem

## (1. Part - if)

Transaction Management

Concurrency control

Locking

**Theorem: A history  $H$  is serializable iff  $SG(H)$  is acyclic**

**Proof:** Let  $T = \{T_1, T_2, \dots, T_n\}$  be the set of transactions over which  $H$  is defined

Let  $T_c = \{T_1, T_2, \dots, T_m\}$  be the set of committed transactions.

The vertices in  $SG(H)$  are the elements of  $T_c$ . Since  $SG(H)$  is acyclic it can be sorted topologically

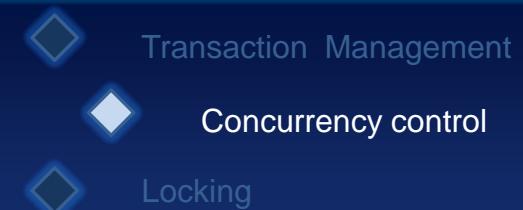
Hypothesis:  $C(H) \equiv H_s$

Let  $p_i$  be in  $T_i$ ,  $q_k$  in  $T_k$  and  $T_i, T_k$  in  $T_c$

$p_i <_H q_k$  are conflicting operations, therefore an edge  $T_i \rightarrow T_k$  exists in  $SG(H)$ .  $T_i$  must appear before  $T_k$  in the topological sort and all the operations of  $T_i$  must appear before those of  $T_k$  in  $H_s$  since conflicting operations are sorted in the same order in  $C(H)$  and  $H_s$ , therefore  $C(H) \equiv H_s$

# Serializability Theorem

## (2. Part - only if)



**Supposition:**  $H$  is SR

Let  $H_s$  be a serial history for which  $C(H) = H_s$

For an edge  $T_i \rightarrow T_k$  in  $SG(H)$  to exist, there must be two **incompatible** operations  $p_i$  and  $q_k$  ordered according to

$$p_i <_{H_s} q_k$$

Since  $C(H) = H_s$  we know that  $p_i <_{H_s} q_k$

Since  $H_s$  is serial and the operation  $p_i$  appears before  $q_k$ , all the operations of  $T_i$  must be ordered before those of  $T_k$  and all of  $T_i$  must appear before  $T_k$

If there was a cycle in  $SG(H)$ , any vertex could appear before itself in the serial history, which is absurd

**SG(H) must be a directed acyclic graph QED**



# Consequences from Proof

- If the complete history  $H$  has an acyclic serialization graph  $\text{SG}(H)$ ,  $H$  is equivalent to some topological sort of  $\text{SG}(H)$
- Since  $\text{SG}(H)$  may have more than one topological sort,  $H$  may have **more than one equivalent serial history**  
**→ serializability is not a deterministic correctness criterion**

# Recoverable Histories

- Serializability guarantees correct concurrent execution of multiple transactions but only in the absence of failures
- Need to formulate correctness criteria that hold also when failures occur
- Must determine if transactions read data from each other.  $T_i$  reads from  $T_k$  in history  $H$  if
  - $w_k[x] < r_i[x]$
  - $a_k !< r_i [x]$
  - if a  $w_m [x]$  exists such that  $w_k [x] < w_m [x] < r_i [x]$ , then  $a_m < r_i [x]$
- $T_i$  reads from  $T_k$  if  $T_k$  wrote  $x$  last and  $T_k$  didn't abort before  $T_i$  read  $x$

# Recoverable Histories (RC)

- A history is ***recoverable (RC)*** if  $T_i$  reads from  $T_k$  ( $i \neq k$ ) in  $H$  and  $c_i$  is in  $H \rightarrow c_k < c_i$
- A history is RC if every transaction in  $H$  commits after the transactions from which it read

$$T_1 = w_1 [x] w_1 [y] w_1 [z] c_1 \quad T_2 = r_2 [u] w_2 [x] r_2 [y] w_2 [y] c_2$$

$$H_7 = w_1 [x] w_1 [y] r_2 [u] w_2 [x] r_2 [y] w_2 [y] c_2 w_1 [z] c_1$$

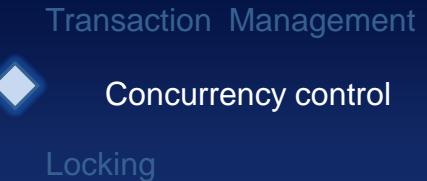
$H_7$  is not RC because  $T_2$  reads  $y$  from  $T_1$  but  $c_2$  appears before  $c_1$

$$H_8 = w_1 [x] w_1 [y] r_2 [u] w_2 [x] r_2 [y] w_2 [y] w_1 [z] c_1 c_2$$

$H_8$  is RC because  $T_2$  commits after  $T_1$  (but is not ACA)



# Histories that Avoid Cascading Abort (ACA)



- A history **avoids cascading aborts (ACA)** if  $T_i$  reads  $x$  from  $T_k$  ( $i \neq k$ ) and  $c_k < r_i[x]$
- A history is ACA if transactions only read from committed transactions

$$T_1 = w_1 [x] w_1 [y] w_1 [z] c_1$$
$$T_2 = r_2 [u] w_2 [x] r_2 [y] w_2 [y] c_2$$
$$H_8 = w_1 [x] w_1 [y] r_2 [u] w_2 [x] r_2 [y] w_2 [y] w_1 [z] c_1 c_2$$

$H_8$  is not ACA since  $T_2$  reads  $y$  from  $T_1$  before  $T_1$  commits

$$H_9 = w_1 [x] w_1 [y] r_2 [u] w_2 [x] w_1 [z] c_1 r_2 [y] w_2 [y] c_2$$

$H_9$  is ACA since  $T_2$  reads  $y$  from  $T_1$  after  $T_1$  commits



# Strict Histories (ST)

- A history is ***strict (ST)*** if

$w_k[x] < o_i[x]$  ( $i \neq k$ ) , either  $a_k < o_i[x]$  or  $c_k < o_i[x]$  ,  
 where  $o_i[x]$  can be either  $r_i[x]$  or  $w_i[x]$

A Transaction only may overwrite data when the transaction that wrote them last finished (committed or aborted)

$T_1 = w_1[x] w_1[y] w_1[z] c_1$

$T_2 = r_2[u] w_2[x] r_2[y] w_2[y] c_2$

$H_9 = w_1[x] w_1[y] r_2[u] w_2[x] w_1[z] c_1 r_2[y] w_2[y] c_2$

$H_9$  is not ST since  $T_2$  overwrites  $x$  before  $T_1$  commits

$H_{10} = w_1[x] w_1[y] r_2[u] w_1[z] c_1 w_2[x] r_2[y] w_2[y] c_2$

$H_{10}$  is ST since  $T_2$  overwrites  $x$  after  $T_1$  commits

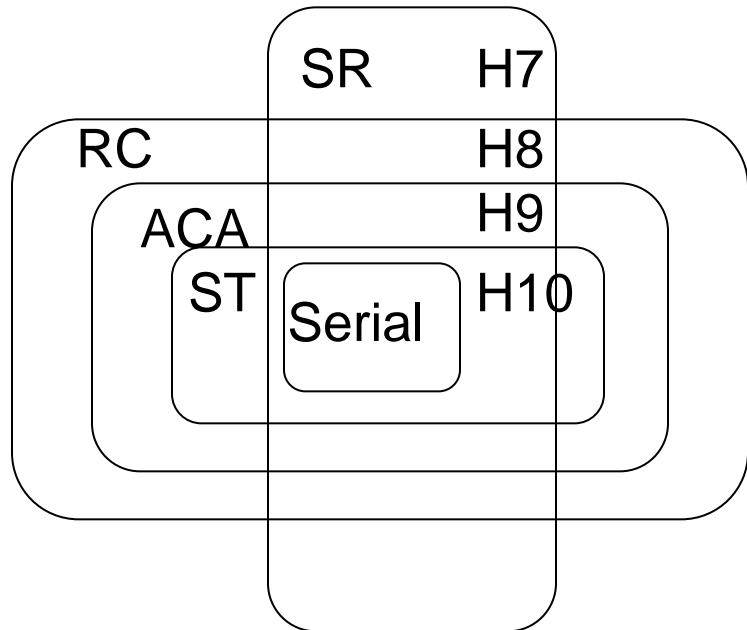
# Relationship of SR, RC, ACA and ST Histories

Transaction Management

Concurrency control

Locking

**Theorem:**  $RC \supseteq ACA \supseteq ST$



- **Proof:**

Let  $H$  be in  $ST$

$T_i$  reads  $x$  from  $T_k$  in  $H$  ( $i \neq k$ )

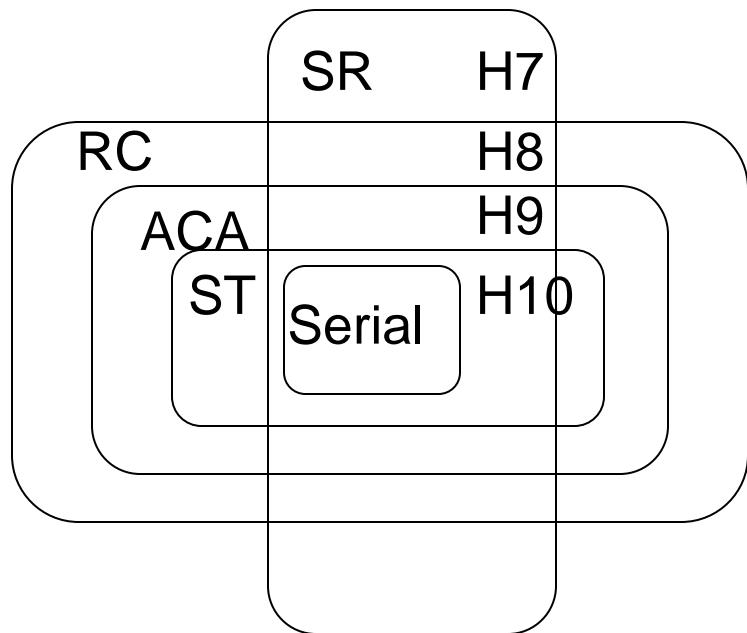
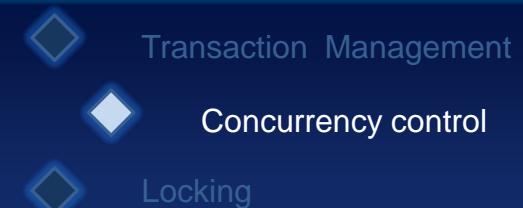
$\rightarrow w_k[x] < r_i[x]$  and  $a_k !< r_i[x]$

By definition of  $ST$ ,  $c_k < r_i[x]$

By definition of  $ACA$ ,  $H$  must also be  $ACA$  and  $ST \supseteq ACA$

Through counterexample ( $H_9$ ) it follows that  $ST \neq ACA$ , therefore  $ACA \supset ST$

# Relationship of SR, RC, ACA and ST Histories



Let  $H$  be in ACA

$T_i$  reads  $x$  from  $T_k$  in  $H$  ( $i \neq k$ ),  
 $c_i \in H$

Because  $H$  is in ACA  $w_k[x] < c_k < r_i[x]$

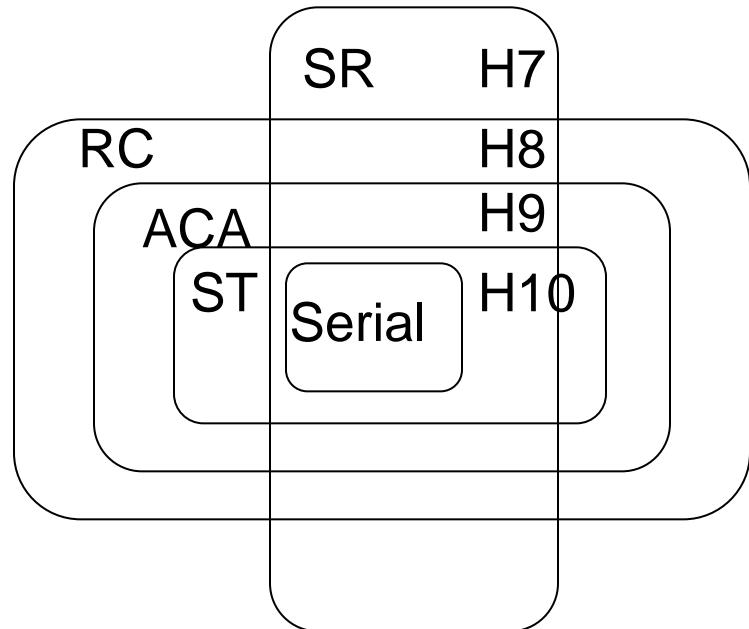
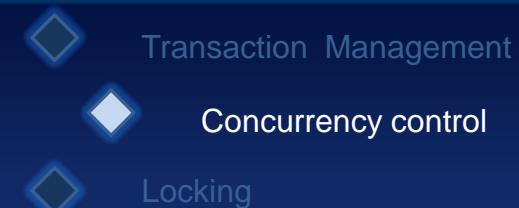
Since  $c_i \in H$ ,  $r_i[x] < c_i$  and  $H$  is RC

$\rightarrow ACA \supseteq RC$

Through counterexample (H8) it follows that  $RC \neq ACA$  and  $RC \supset ACA$



# Relationship of SR, RC, ACA and ST Histories



- It is possible to show that there exist histories that are in ST (and by previous proof in ACA and RC) that are not in SR. Similarly, there is at least one history (H7) which is in SR but not in RC
  - SR is not comparable with ST, ACA and RC

For a good explanation of this basic treatment read Bernstein et al.  
For further proofs read Vossen or the newer Weikum and Vossen

# Prefix Commit Closedness

- A History is ***prefix commit closed*** with respect to a property when all possible prefix histories of the normally terminated transactions have the same property (e.g. SR) as the full history,  
i.e.
- Prefix commit closedness is required in order to guarantee correctness in the case of system failure.

# View Equivalence

- **Equivalence** can be defined based on effects of a history (as opposed to analyzing the conflicts)
- We don't know anything about the values written by  $w[x]$  but we know that there is a function that reads values and produces a write operation
- → if each transaction's reads read the same values in a history, the writes produced must be the same in both histories
- → if the final write on  $x$  is the same in both histories, then the final value of  $x$  must be the same in both histories

# View Equivalence

The final write of  $x$  in a History  $H$  is  $w_i[x] \in H$ , such that  $a_i \notin H$  and for any  $w_k[x] \in H$  ( $i \neq k$ ) either  $w_k[x] < w_i[x]$  or  $a_k \in H$

**Two histories  $H$  and  $H'$  are view equivalent if:**

- 1) They are over the same set of transactions and have same operations
- 2) For any  $T_i, T_k$  such that  $a_i, a_k \notin H$  (hence  $a_i, a_k \notin H'$ ) and for any  $x$ , if  $T_i$  reads  $x$  from  $T_k$  in  $H$  then  $T_i$  reads  $x$  from  $T_k$  in  $H'$
- 3) For each  $x$  if  $w_i[x]$  is the final write of  $x$  in  $H$  it is also the final write of  $x$  in  $H'$

# View Serializability

- A history is **view serializable** if for any prefix  $H'$  of  $H$ ,  $C(H')$  is view equivalent to some serial history

**NOTE:** if committed projection of every prefix is view equivalent to a serial history we ensure that view serializability is a commit closed property

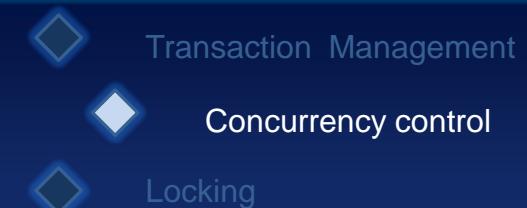
$$H_{12} = w_1 [x] w_2 [x] w_2 [y] c_2 w_1 [y] c_1 w_3 [x] w_3 [y] c_3$$

$C(H_{12}) = H_{12}$  is view equivalent to  $T_1 T_2 T_3$  (same  $w_3 [x] w_3 [y]$  )

for the prefix  $H'_{12} = w_1 [x] w_2 [x] w_2 [y] c_2 w_1 [y] c_1$

$C(H'_{12})$  is not view equivalent to either  $T_1 T_2$  nor to  $T_2 T_1$

# View Serializability vs. Conflict Serializability



**Theorem:** If  $H$  is conflict serializable it is also view serializable. The converse is not, generally, true.

**Proof:**

Suppose  $H$  is conflict serializable. The arbitrary prefix  $H'$  has  $C(H')$  that is equivalent to some serial history  $H_s$

Hypothesis:  $C(H')$  is view equivalent to  $H_s$

$C(H')$  and  $H_s$  are defined over the same operations since they are conflict equivalent.

Must show that a) they read the same values and b) have same final writes for all data values



# Proof VSR ⊢ CSR (cont.)

Suppose  $T_i$  reads  $x$  from  $T_k$  in  $C(H')$  →

$w_k[x] <_{C(H')} r_i[x]$  and there is no  $w_m[x]$  such that  $w_k[x] <_{C(H')} w_m[x] <_{C(H')} r_i[x]$

Because  $w_k[x]$  and  $r_i[x]$  conflict with each other and both conflict with  $w_m[x]$  and because  $H_s$  and  $C(H')$  order conflicting operations the same way (conflict SR) →  $w_k[x] <_{C(H_s)} r_i[x]$  and there is no intermediate write operation

Therefore,  $T_i$  reads  $x$  from  $T_k$  in  $H_s$  and  $T_i$  reads  $x$  from  $T_k$  in  $C(H')$  and they have the same read-from relationships. Since CSR orders conflicting operations the same way, the last write must be the same

$C(H')$  and  $H_s$  are view equivalent and since  $H'$  is arbitrary prefix they are view serializable

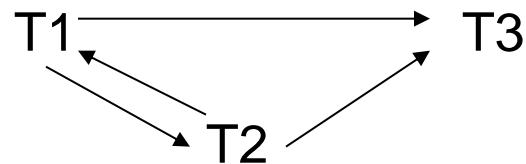
# Proof VSR $\supset$ CSR (cont.)

**2. Part:** show converse is not, generally, true by counterexample

$$H_{13} = w_1 [x] w_2 [x] w_2 [y] c_2 w_1 [y] w_3 [x] w_3 [y] c_3 w_1 [z] c_1$$

$H_{13}$  is view serializable. Consider any prefix  $H_{13}'$ . If it includes  $c_1$  it is equivalent to  $T_1 T_2 T_3$ , if it includes up to  $c_3$  it is equivalent to  $T_2 T_3$ , if it includes up to  $c_2$  it is equivalent to  $T_2$

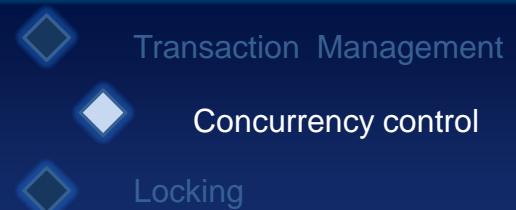
$H_{13}$  is not CSR because of cycle in serialization graph



# CSR vs. VSR

- While **VSR** is more inclusive, i.e. more schedules are **VSR** than **CSR**, for practical reasons **CSR** is preferred
- **CSR** is easily implementable
- A scheduler based on **VSR** would have to produce and test all possible histories efficiently (NP complete)
  
- Will concentrate on how to implement **CSR**

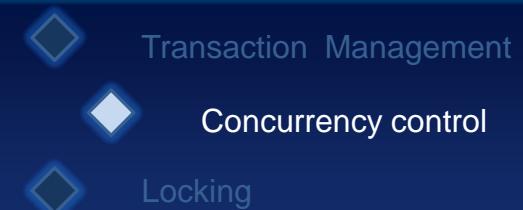
# Schedulers



- **Correctness Criterion:** Conflict serializability
  - practical
  - efficient
  - easy to implement
- **Different implementations possible based on**
  - locking
  - time stamping
  - serialization graph testing



# Aggressive vs. Conservative Schedulers



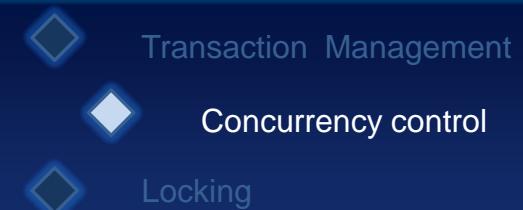
- Aggressive schedulers pass-on command immediately, must detect and resolve conflicts
- Conservative schedulers hold back commands until they are certain that no deadlock will occur (serial execution, predeclaration - query analysis and relation-level locks)
- **Trade-off:**
  - few conflicts, cheap rollback - use aggressive scheduler
  - many conflicts, rollback expensive (or not feasible) - use conservative scheduler



# Locks

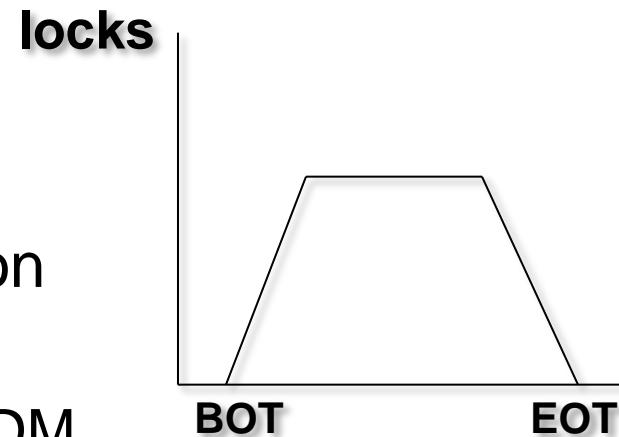
- **Locks** are associated with data objects and are requested by the scheduler for a given transaction. Scheduler is responsible for lock management, i.e. locking sequence
- **Locks** can be viewed as triplet [data item, lock type, TX]
- Operation determines the **lock type**
- **Lock** conflicts correspond to conflicts between operations
  - two locks  $pl_i[x]$  and  $ql_k[y]$  are incompatible if  $i \neq k$ ,  $x = y$  and  $p,q$  are incompatible operations
- **Notation:**
  - $pl_i[x]$  - operation p in transaction i locks object x
  - $pu_i[x]$  - operation p in transaction i unlocks x
  - $rl_i[x], ru_i[x], wl_i[x], wu_i[x]$  basic read and write locks

# Two Phase Locking Protocol (2PL)

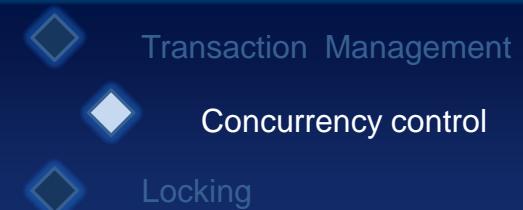


## Basic 2PL

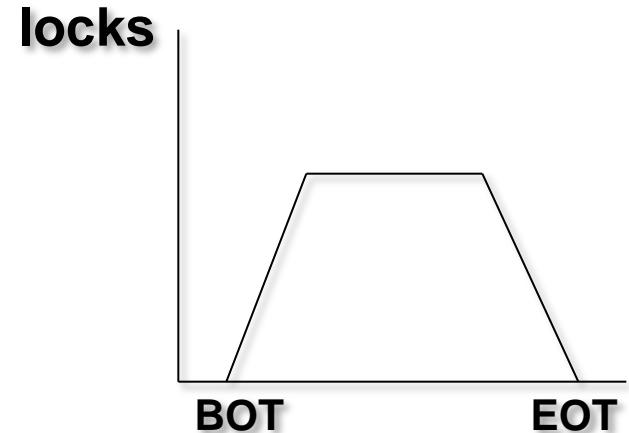
- 1) Scheduler gets command from transaction manager (TM), checks if lock is set
  - lock is set (owned)** - command directly to DM
  - lock not set** - check if object unlocked
    - yes** - set lock and pass command
    - no** - check compatibility
      - compatible** - increase reference count, pass cmd.
      - incompatible** - wait



# Two Phase Locking Protocol (2PL) (cont.)



- 2) The earliest point at which a lock can be released is after the DM has acknowledged that the operation was carried out (handshake principle)
- 3) If a transaction  $T_i$  released a lock,  $T_i$  may not acquire any new locks



# 2PL Example

 $T_1 = r_1[x] \rightarrow w_1[y] \rightarrow c_1$  $T_2 = w_2[x] \rightarrow w_2[y] \rightarrow c_2$  $H_1 = rl_1[x] \ r_1[x] \ ru_1[x] \ wl_2[x] \ w_2[x] \ wl_2[y] \ w_2[y] \ wu_2[x] \ wu_2[y] \ c_2$   
 $wl_1[y] \ w_1[y] \ wu_1[y] \ c_1$ 

This history ***violates 2PL*** and is ***not serializable***

$T_1$  unlocks x,  $T_2$  can lock x ;  $T_2$  unlocks y,  $T_1$  can lock y → cycle in SG

# 2PL Example (cont.)

## Execution observing 2PL:

initially no locks set;  $r_1[x]$  from TM to scheduler,  $rl_1[x]$  set,  $r_1[x]$  to DM with ack

$w_2[x]$  from TM to scheduler,  $wl_2[x]$  cannot be set,  $w_2[x]$  is delayed and queued

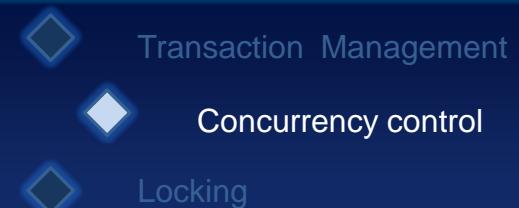
$w_1[y]$  from TM to scheduler,  $wl_1[y]$  set,  $w_1[y]$  to DM with ack from DM  
 $c_1$  from TM to scheduler,  $c_1$  to DM, DM carries out  $c_1$  and sends ack to scheduler

scheduler releases locks  $ru_1[x]$  and  $wu_1[y]$  since no further ops. required by  $T_1$

scheduler takes  $w_2[x]$  from queue, sets  $wl_2[x]$ ,  $w_2[x]$  to DM with ack from DM

$w_2[y]$  from TM to scheduler,  $wl_2[y]$  set,  $w_2[y]$  to DM with ack from DM  
 $c_2$  from TM to scheduler,  $c_2$  to DM, DM executes  $c_2$  and acks,  
scheduler releases  $w_2[x]$ ,  $w_2[y]$  → H is serializable (in this particular case it is also serial)

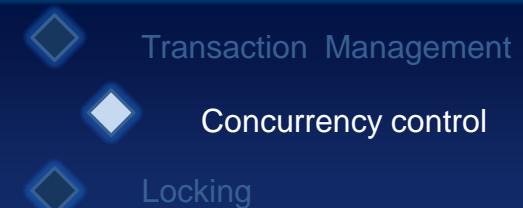
# Correctness of a 2PL Scheduler



- Want to prove that a scheduler obeying the basic **2PL** produces conflict serializable schedules
- **Proof outline:**
  - all histories produced by the scheduler must be provably *SR*
    - characterization of all possible histories
    - proof that these histories are *SR*, i.e. show that  $SG(H)$  is acyclic



# Proof of Correctness of 2PL Schedulers



## Characterization of possible histories:

Order of requested locks reflects order of execution (hand-shake) and of incompatible operations

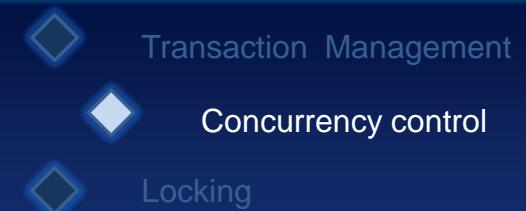
From the definition of **2PL** (3 rules) it follows that:

$ol_i[x] < o_i[x]$  lock is always requested before operation

$o_i[x] < ou_i[x]$  operation is always executed before releasing lock



# Proof of Correctness of 2PL Schedulers (cont.)



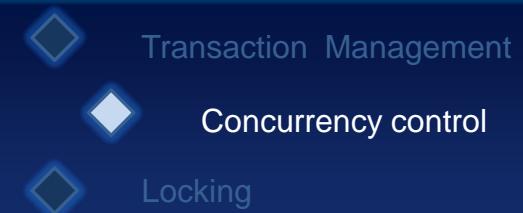
## Proposition 1:

Let  $H$  be a history produced by a 2PL scheduler. If  $o_i[x]$  is in  $C(H)$  it follows that  $ol_i[x] < o_i[x] < ou_i[x]$

In case of 2 incompatible ops., only one can hold the lock, therefore  $pu_i[x] < ql_k[x]$  or  $qu_k[x] < pl_i[x]$



# Proof of Correctness of 2PL Schedulers (cont.)



## Proposition 2:

For every  $H$  in  $C(H)$  it follows that  $pu_i[x] < ql_k[x]$  or  $qu_k[x] < pl_i[x]$

A transaction can't request additional locks once it released the first lock (Rule 3), therefore  $pl_i[x] < qu_i[y]$

This means that in a history all lock operations of a transaction must precede the first unlock operation



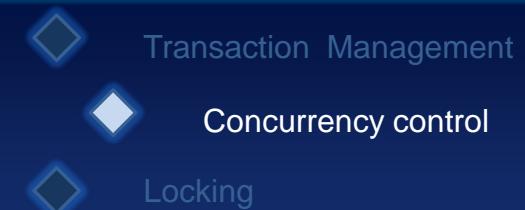
# Proof of Correctness of 2PL Schedulers (cont.)

## Proposition 3:

Let  $H$  be a history produced by a 2PL scheduler. If  $p_i[x]$  and  $q_i[y]$  are in  $C(H)$ , then  $pl_i[x] < qu_i[y]$

Using the above properties we must show that every 2PL history has an acyclic graph (proof in 3 steps)

# Proof of Correctness of 2PL Schedulers (cont.)



**Hypothesis:** Every 2PL schedule has an acyclic SG

$SG(H)$  contains nodes only for committed transactions in  $H$

If  $T_i \rightarrow T_k$  is in  $SG(H)$  then some  $o[x]$  of  $T_i$  executed before  $T_k$  and conflicted with an  $o[x]$  in  $T_k$ . Therefore,  $T_i$  must have released a lock on  $x$  before  $T_k$  could set the lock on  $x$

Suppose a path  $T_i \rightarrow T_k \rightarrow T_m$  exists in  $SG(H)$ .  $T_i$  set and released some lock before  $T_k$  and  $T_k$  before  $T_m$ . By the two phase rule,  $T_i$  set all its locks before it released any lock.  $T_i$  released some lock before  $T_k$  could acquire some lock. By transitivity the same holds for  $T_m$



# Proof of Correctness of 2PL Schedulers (cont.)



By induction, in a path  $T_1 \rightarrow \dots \rightarrow T_n$ ,  $T_1$  released a lock before  $T_n$  set some lock.

**Supposition:**  $SG(H)$  has a cycle  $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$

By previous step,  $T_1$  released a lock before  $T_1$  set a lock

This contradicts the two phase rule which states that no transaction may acquire a lock after releasing any lock

Therefore, a cycle cannot exist and the Serializability Theorem implies that  $H$  is SR

→ Every schedule produced by a 2PL scheduler is SR



# Deadlocks

- 2PL schedulers are ***not deadlock-free***

- Deadlocks result from

- normal execution according to the 2PL rule
  - upgrading of locks (from read-lock to write-lock)

- Example of deadlock in 2PL execution

$$T_1 = r_1 [x] \rightarrow w_1 [y] \rightarrow c_1 \quad T_3 = w_3 [y] \rightarrow w_3 [x] \rightarrow c_3$$

No locks set initially

Scheduler receives  $r_1[x]$ , sets  $rl_1[x]$ , passes  $r_1[x]$  to DM

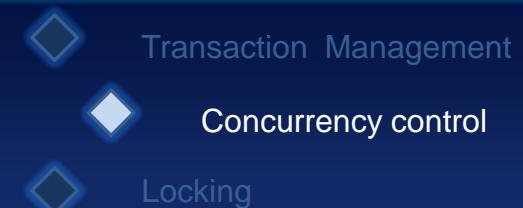
Scheduler receives  $w_3[y]$ , sets  $wl_3[y]$ , passes  $w_3[y]$  to DM

Scheduler receives  $w_3[x]$ , can't set  $wl_3[x]$ , blocks  $w_3[x]$

Scheduler receives  $w_1[y]$ , can't set  $wl_1[y]$ , blocks  $w_1[y]$

→ **deadlock!** Scheduler can't pass on an operation without a lock  
 but can't set lock because these are already set. By 2PL rule  
 scheduler can't release a lock and request a new lock for a  
 transaction

# Deadlocks through Lock Escalation



$T_4 = r_4[x] \rightarrow w_4[x] \rightarrow c_4$

$T_5 = r_5[x] \rightarrow w_5[x] \rightarrow c_5$

## No locks set initially

Scheduler receives  $r_4[x]$ , sets  $rl_4[x]$ , passes  $r_4[x]$  to DM

Scheduler receives  $r_5[x]$ , sets  $rl_5[x]$ , passes  $r_5[x]$  to DM

Scheduler receives  $w_4[x]$ , can't set  $wl_4[x]$ , blocks  $w_4[x]$

Scheduler receives  $w_5[x]$ , can't set  $wl_5[x]$ , blocks  $w_5[x]$

→ deadlock!



# Dealing with Deadlocks

- **Deadlocks** are eliminated by releasing a resource = releasing a lock
- **Locks** can only be released in case of deadlock by aborting a transaction
- **Detection of deadlocks**
  - timeout
  - Wait-for Graph (WFG)

# Timeouts

- **Passive mechanism** to deal with deadlock
- Scheduler aborts a transaction that waits ***too long*** for lock
- Practical, easy to implement
- Correct, but non-deadlocked transactions may be accidentally aborted → unnecessary delay of a TX
- **Main problem:** determining the right timeout interval
  - **long interval** - deadlocked transactions wait too long before deadlock is detected/resolved
  - **short interval** - transactions are unnecessarily aborted and rolled back, sensitive to overload

# Wait For Graphs (WFG)

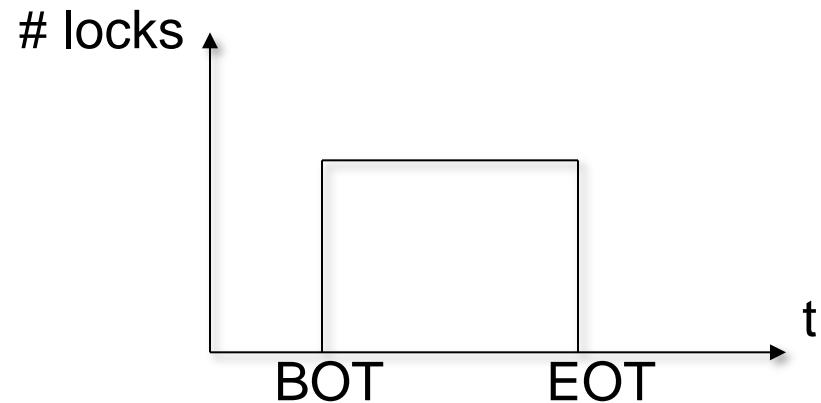
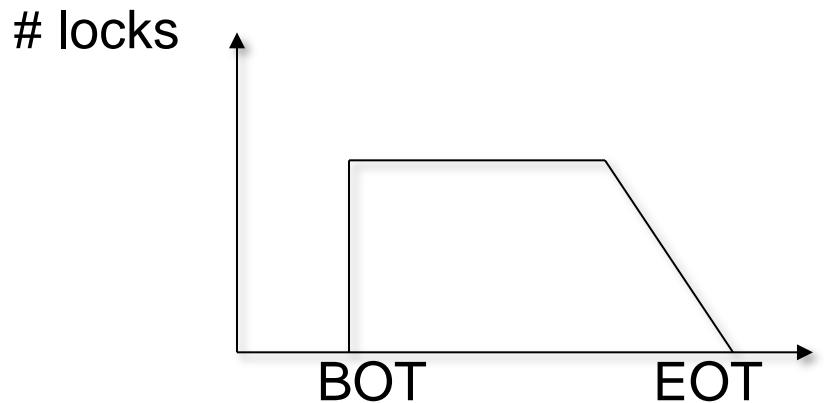
- **Active mechanism** - scheduler determines deadlocked transactions
- **WFG** has an edge for each transaction that waits for another transaction
- Deadlock exists when **WFG is cyclic**
- Scheduler must maintain **WFG**
  - each blocked lock request requires an edge in **WFG**
  - when last lock of a blocking transaction is released, edge in **WFG** must be deleted
- What kind of support mechanisms?
- When to test for cycles: after each new edge, after  $n$  entries, after predetermined time?

# Selecting a Victim

- **Criteria for selecting a victim for roll-back**
  - minimization of lost work
  - minimization of roll-back costs
  - priority for transactions that must finish
  - maximization of broken-up cycles
  - Example: Sybase chooses transaction that has consumed the least CPU-time
- **Avoiding starvation**
  - transactions may participate repeatedly in cycles
  - selection algorithm must be fair

# Conservative 2PL - Preclaiming

- Transaction Management
- Concurrency control
- Locking



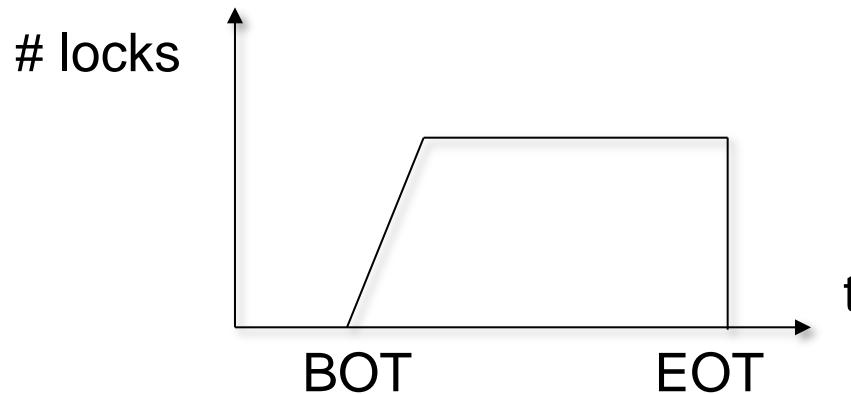
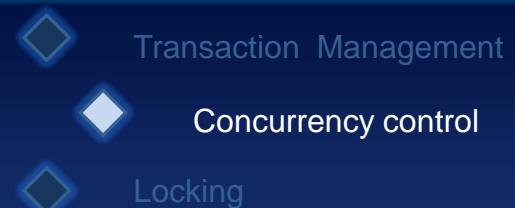
- Locks are requested and granted before execution begins
- If scheduler can't set a lock, the requests are queued
- Whenever another transaction finishes it is attempted to complete a waiting transaction's requests
- Conservative 2PL refers to acquisition of locks**, not release, i.e. release may be gradual or all at EOT (strict preclaiming 2PL)



# Conservative 2PL - Preclaiming

- **Preclaiming is guaranteed deadlock-free**
- Since resources are acquired on an all or nothing basis, there can't be an edge in the WFG → **no deadlock**
- Since conflicts are avoided, there is *no need for rollback*
  - **positive** - no unnecessary work is done, resources are optimally used
  - **negative** - individual transactions may be delayed, response time suffers
  - since read and write sets are difficult to anticipate in generic queries (can only determine at type-level, i.e. relation, with syntactic analysis), useful in main memory DBMSs and real-time systems

# Strict 2PL



- **Locks are released after commit (or abort)**
- **Commit and abort** are the first point in a transaction where the system can guarantee (in the absence of additional information) that no new operations and lock requests will be submitted by a transaction
- Strict refers to the *release* of the locks (the way they are requested is irrelevant as long as they obey the 2PL rule)

# Implementation Aspects - Lock Manager



- **Lock manager** manages the lock table
  - **lock**(transaction-id, oid, mode, ptr)      insert in lock table
  - **unlock**(transaction-id, oid)                delete from lock table
- Lock manager must be **efficient**
- **Lock table** is hash table keyed on oid
  - objects may be pages, records, etc. (depending on DBMS)
  - lock table entry consists of oid, #ofTX holding lock, lock-type, ptr. to queue of lock requests
- Transaction manager also keeps transaction table with pointer for each transaction entry to a list of locks held by the transaction
- Ops. on lock table and transaction table must be atomic



# Implementing Lock and Unlock Requests



- **If shared lock requested**
  - Q empty, object unlocked → grant lock, update lock table
  - Q empty, object locked shared → grant lock, increment TX ctr.
  - Q empty, object locked exclusive → add request to Q, block TX
- **If exclusive lock requested**
  - Q empty, object unlocked → grant lock, update lock table
  - Q empty or not, object locked shared/exclusive → add request to Q, block TX



# Implementing Lock and Unlock Requests(cont.)



- When transaction *commits or aborts*, use list of locks held by transaction, get corresponding entry in lock table, check request queue for oid, if not empty take request(s) at head of queue, update lock table entry and wake up blocked TX
- Request queues are FIFO to *avoid starvation*



# Lock Upgrades, Convoys, Latches



- When transaction holds shared lock and wants **exclusive lock**,
  - it is either granted immediately (if no other transaction has s-lock)
  - it is inserted at head of request queue (if already locked shared) to avoid deadlocks due to lock upgrades
- **Convoys** are formed when a transaction holding a lock on a heavily requested object is suspended by the OS
  - TX holding lock is blocked and blocks other TXs waiting for lock
  - once a convoy forms it tends to be stable
  - convoys are one of the disadvantages of implementing DBMS on top of OS with standard preemptive scheduling
- **Latches** are short-duration locks set only for the duration of a physical I/O operation to guarantee atomicity



# Index Locks



- **B+ trees** (or variants thereof) are the most popular index mechanisms in today's DBMSs
- Locking B+ indexes with 2PL causes possibly *long delays*
  - index is dynamic, i.e. insertion of an index entry may cause rebalancing and splitting all the way to the root
  - the whole path from root to leaf would have to be exclusively locked
  - this would degrade the index to single user mode
  - locks would have to be held till end of transaction
- Need to find **non-2PL protocols** for tree indexes that are serializable



# Locks on B-trees and its Variants



Bayer, R., Schkolnick, M.: *Concurrency of Operations on B-Trees*, Acta Informatica, 9,1, pp 1-21

Kwong, Y., Wood, D.: *A New Method for Concurrency in B-Trees*, IEEE Trans. On SW Eng., 8,3, May, 1982

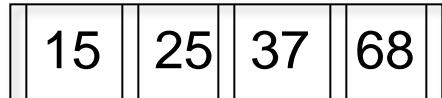
Mohan, C.: *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiactions Operating on B-Tree Indexes*, IBM RJ7008, 9/6/89

- **Goal:** to find algorithms that allow concurrent operations on B-Trees
- **Problem:** through dynamic reorganizations the nodes may be split all the way to the root
- **Approach:** identify „safe“ nodes that allow the locking of parts of the path
- **Safe nodes in B-Trees:** A node with  $k \leq \mu \leq 2k$  is safe if at insert  $\mu < 2k$  and at delete  $\mu > k$
- A safe node can't produce overflows or underflows and isolates the nodes above and below



# B-Tree Locking Algorithms

- Concurrency control
- Locking
- Isolation levels



Safe for delete, unsafe for insert

Base Algorithm (Bayer and Schkolnick)



Safe for insert, unsafe for delete

There exist 2 types of locks: read locks  $\rho$ , exclusive locks  $\xi$

Scheduler assigns locks in FIFO order

```
READ: lock root with  $\rho$  lock
      make root current
      while current != leaf
      begin
          lock son with  $\rho$  lock
          release  $\rho$  on current
          make locked son current
      end
```



# B-Tree Locking Algorithms (cont.)

- Concurrency control
- Locking
- Isolation levels

## UPDATE:

```
lock root with ξ lock
get root and make root the current
while current != leaf do
```

```
begin
```

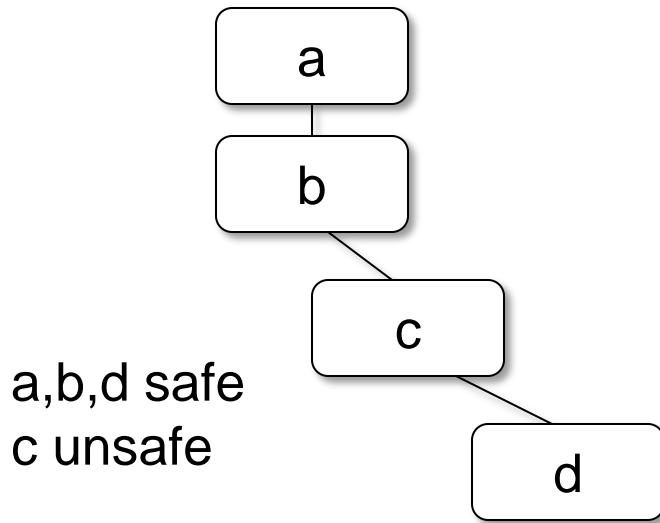
```
    lock son with ξ lock
    make locked son the current
    if current safe then release ξ lock on parent(s)
```

```
end
```



# B-Tree Locking Algorithms (cont.)

- Concurrency control
- Locking
- Isolation levels



## Example:

set  $\xi$  lock on a, set  $\xi$  lock on b,  
since b is safe, release  $\xi$  lock on a  
set  $\xi$  lock on c, c is unsafe, hold  $\xi$  lock on b  
set  $\xi$  lock on d, d is safe, release  $\xi$  locks  
on b and c

# B-Tree Locking Algorithms (cont.)



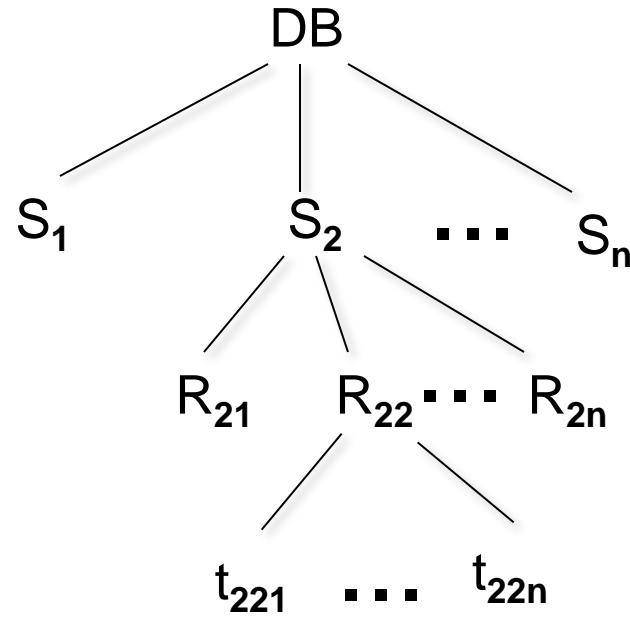
- **Alternative:** updater requests shared lock, only if node in which insert is to occur is unsafe
  - a) descent through index is repeated with previous algorithm, or
  - b) lock upgrade to parent is requested
- **Lock upgrades may cause deadlocks**
- Normal index locking in tree structured indexes is *deadlock free* (unsafe nodes are by default locked in exclusive mode)
- Many other refinements are possible and quite profitable given the use an index gets



# Multigranularity Locking

- Setting and managing locks is *expensive*
- *Transactions* accessing whole relations pay a high overhead if locks are set at tuple level
- Want to achieve ***balance*** between low granularity (= higher parallelism) and lock management costs
  - large transactions accessing many records set locks on big granules, small transactions set locks on small granules → higher parallelism but moderate lock management costs
- Database resources should be organized *hierarchically*

# Multigranularity Locking (cont.)



- Resources must be locked in *hierarchical order*
- To lock  $t_{221}$  we must lock  $R_{22}$ ,  $S_2$  and DB
- With only two available lock modes (exclusive and shared) this would result in single user operation when a write operation sets an exclusive lock, only readers could execute concurrently
- **Need new lock modes**

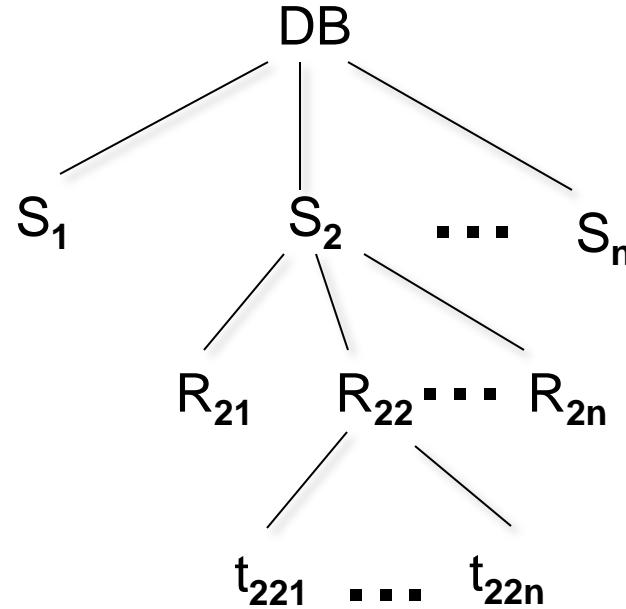


# Lock Modes

	IS	IX	S	SIX	X
IS	+	+	+	+	-
IX	+	+	-	-	-
S	+	-	+	-	-
SIX	+	-	-	-	-
X	-	-	-	-	-

- Intention locks result in **5 lock modes instead of 2**
- ir IS ***intention to read*** lower objects that may be locked IS or S
- iw IX ***intention to write*** lower object that may be locked in any mode
- r S ***read*** lock on node and all its successors
- w X ***write*** lock, allows exclusive access to node and successors
- riw SIX allows read access to node and declares intention to modify successors. These may be locked in X, SIX or IX modes

# Example Multigranularity Locking

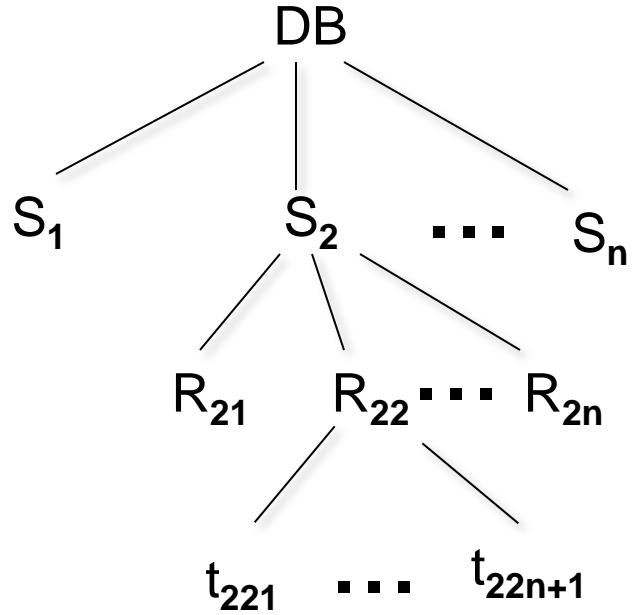


- Concurrency control
- Locking
- Isolation levels

- Exclusive (write) lock on  $t_{225}$  requires
- IX lock of DB (avoids S or X lock of other transaction on whole DB)
- IX lock on  $S_2$
- IX lock on  $R_{22}$
- X lock on  $t_{225}$

# Example 2 Multigranularity Locking

- Concurrency control
- Locking
- Isolation levels



- Insert  $t_{22n+1}$
- IX lock of DB (avoids S or X lock of other transaction on whole DB)
- IX lock on  $S_2$
- X lock on  $R_{22}$

# Hierarchical Locking

- **Hierarchical locks** must be set from the root to the leaves and released from the leaves to the root
  - this avoids that a lock is set at a lower level while the upper level has been unlocked
- **Hierarchical locks** alone do not guarantee serializability, must use them together with 2PL protocol
  - 2PL determines *when* to lock/unlock objects
  - MGL indicates *how* to lock objects of different granularity
- Lock manager only knows that a new (extended) conflict matrix exists

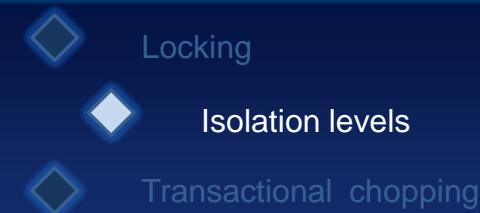
# Performance of Locking Mechanisms



- **Performance of lock mechanisms** is difficult to determine
- Performance analysis has been done only for ideal and highly simplified conditions
- Competition for resources (memory, CPU, I/O channels, data) leads to ***thrashing***
- Data contention can lead to thrashing even in an *idealized case* (unlimited main memory) because of
  - rollback of transactions (e.g. because of deadlocks)
  - blocking of transactions (e.g. because of lock conflicts)
- Measurements show that ***blocking*** is the *most important* factor until the thrashing point is reached



# Thrashing of Locking Mechanisms



- **Thrashing** may occur with only 2% of rolled back transactions
- Waiting time is usually longer than rollback time
- In thrashing mode, every new transaction usually results in multiple blocked transactions
- Once the thrashing point is reached, deadlocks dominate
- **Rule of thumb:** thrashing occurs at the latest when 50% of transactions are blocked for resources



# Workload Limits in Lock-based Systems

Locking

Isolation levels

Transactional chopping

- A DBMS should never be operated above the thrashing point  
→ must be able to predict or at least analyze thrashing
- **Data contention workload**

$$W = k^2N/D \quad \text{where}$$

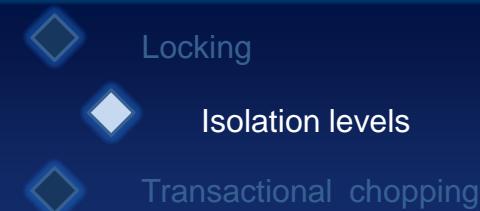
$k$  = number of locks

$N$  = number of active TXs

$D$  = # of data objects  
(locking units)

- From performance model and simulation thrashing begins at  $W = 1.5 \rightarrow k^2N/D < 1.5$
- $W = 1.5$  is upper limit since hot spots lower performance
- Number of locks **dominates** → use short transactions, break up long transactions if possible

# Effect of Shared Locks and Hot Spots on Performance



- If a fraction of the locks are ***shared locks***, the workload that leads to thrashing improves

$$W_s = (1 - s^2) k^2 N/D$$

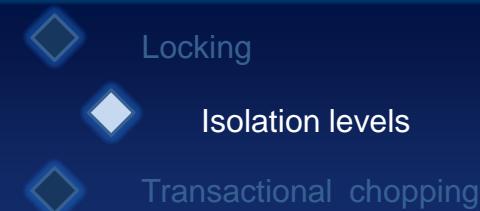
- Skewed data access distribution (***hot spots***) reduces the thrashing threshold

$$W_h = (1 + (q - p)^2/p(1-p)) k^2 N/D$$

where  $p$  is the fraction of frequently used data and  $q$  is the probability that a data access will be on those data

e.g.:  $p=0.2$   $q=0.8$  means that 80% of accesses occur on 20% of the data



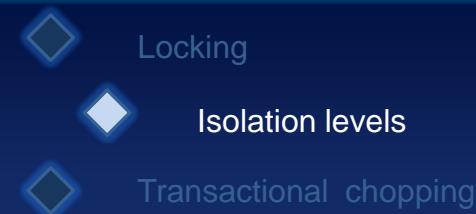


# Levels of Isolation

- Commercial DBMSs and SQL provide levels of isolation
- The **isolation level controls** the extent to which a given transaction is exposed to actions of concurrent transactions
- There are **4 levels of isolation** from which the user may choose:
  - *read uncommitted*
  - *read committed*
  - *repeatable read*
  - *serializable*
- By choosing the isolation level the user may *trade-off* greater concurrency at the expense of exposure to uncommitted changes



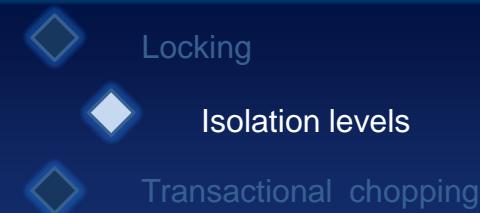
# Serializable



- The SQL ***serializable isolation level*** corresponds to the strict serializable correctness we discussed
- Locks are acquired according to 2PL and released at commit
- Serializable additionally uses ***index locking***
- Index locking avoids phantoms

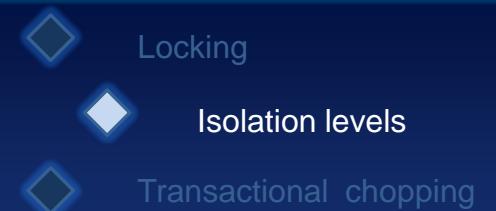


# Repeatable Read



- **Repeatable read** ensures that a transaction reads only changes made by committed transactions and that no value read or written by T is changed by any other transaction until T is complete
- **Repeatable read** uses same locking strategy as serializable except that it doesn't lock indexes
- Because *no index locking* is used, phantoms may appear (i.e. records inserted by another concurrent transaction may or may not be visible)





# Read Committed

- **Read committed** ensures that a transaction  $T$  reads only changes made by committed transactions and that no value **written** by  $T$  is changed until  $T$  is complete.
- A value **read** by  $T$  may be changed by another transaction while  $T$  is still in progress
- $T$  is exposed to phantoms
- Exclusive locks are acquired before writing and held to the end, shared locks are acquired before reading but released (only ensures that it reads committed values)





# Read Uncommitted

- **Read uncommitted** allows a transaction to read changes made to an object by an ongoing transaction.
- The object can be changed further while  $T$  is in progress
- $T$  also sees phantoms



# Transaction Chopping

- Isolation levels
- Transactional chopping
- Non-locking schedulers

From the data contention workload

$$W = k^2 N/D \quad \text{where} \quad \begin{aligned} k &= \text{number of locks} \\ N &= \text{number of active TXs} \\ D &= \text{number of data objects} \\ &\quad (\text{locking units}) \end{aligned}$$

we infer that keeping small the number of locks being held in a database will *reduce the potential for conflict*

A key to reducing the number of locks being held simultaneously is to *shorten the transactions* setting them

**Basic question is then:** how can we shorten the transactions and maintain correct semantics?

# TX Chopping: Application Scenario

Isolation levels

Transactional chopping

Non-locking schedulers

A credit card company allows its customers to charge certain amounts without prior verification

At night an update transaction inspects each customer's account that was charged against during the day, updates the account's balance and updates the issuing bank's total

The update transaction is long running and due to the nature of credit card charges (24x7) the company wants to move away from running the update TX only at night



# TX Chopping: Application Scenario(2)

Isolation levels

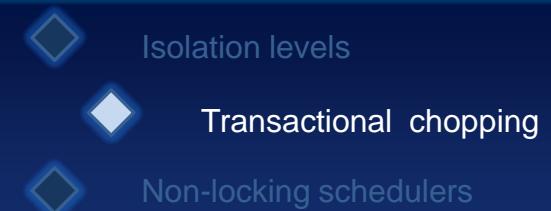
Transactional chopping

Non-locking schedulers

- **The single long update transaction is replaced by many small ones**
  - each small TX updates a customer's account and updates the issuing bank's total
  - readers accessing individual accounts will see no difference whether the small TXs or the single large TX updated the accounts
  - small TXs may now collide when updating the issuing bank's total → break small transactions into one that updates the account and one that updates the bank total
- *Why is this correct?*
  - all small transactions must be executed
  - no transactions accessing both the customer accounts and the bank total simultaneously exist (can't get inconsistent view)



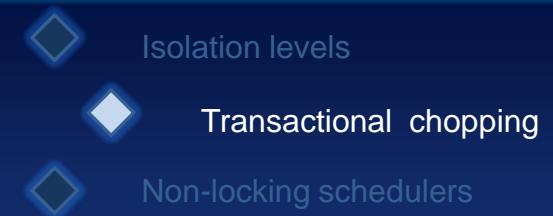
# A Model for Transaction Chopping



- Transaction chopping *depends* on the mix of transactions
- Assume that this mix is *known*
  - first cast everything into parameter-less SQL calls
  - simple `SELECT` and `UPDATE` (can be treated as r, w)
  - later will introduce/consider programs with richer control flow



# Credit Card Scenario revisited



## Three transaction classes:

TX updating a customer account and the issuing bank's total Accts. A1,A2,A3 belong to B1, A4,A5 to B2

$$t1 = r1(A1)w1(A1)r1(B1)w1(B1)$$

$$t2 = r2(A4)w2(A4)r2(B2)w2(B2)$$

TX reading a customer's account balance

$$t3 = r3(A2)$$

TX comparing sum of individual balances and sum of bank totals

$$t6 = r6(A1)r6(A2)r6(A3)r6(A4)r6(B1)r6(B2)$$



# Credit Card Scenario continued

-  Isolation levels
-  Transactional chopping
-  Non-locking schedulers

Previous transactions really consist of SQL statements:

$$t1 = r1(A1)w1(A1)r1(B1)w1(B1)$$

```
SELECT Balance INTO :oldbalance FROM Accounts
```

```
    WHERE AccountNo = A1;
```

```
UPDATE Accounts SET Balance = :newbalance
```

```
    WHERE AccountNo = A1;
```

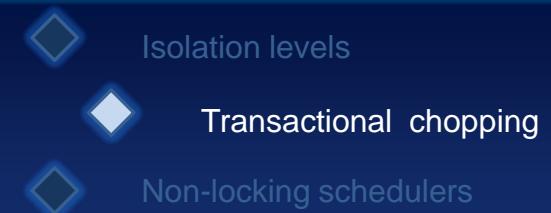
```
SELECT Total INTO :oldtotal FROM Banks
```

```
    WHERE BankNo = B1;
```

```
UPDATE Banks SET Total = :newtotal
```

```
    WHERE BankNo = B1;
```





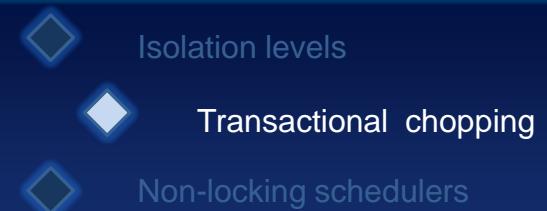
# Transaction Chopping

## Definition:

Let  $T$  be a transaction program. A chopping of  $T$  is a decomposition of  $T$  into pieces  $T_1, T_2, \dots, T_k$  ( $k \geq 1$ ) such that each database operation of  $T$  is performed in exactly one piece of  $T$  and the order of invocation is preserved



# Guaranteeing Execution



If a piece resulting from chopping is aborted due to deadlock → *repeat until it commits*

If a piece is aborted due to program-initiated rollback (e.g. explicit constraint check)

- no other (later) piece of the transaction may execute
- only the first piece may contain a program-initiated rollback statement

**Execution order can be preserved through transactional message queues**



# How to guarantee correctness: Chopping Graph

Isolation levels

Transactional chopping

Non-locking schedulers

- Given a set of transactions to be chopped, we want to know whether a given chopping preserves serializability of the set of original transaction programs

Given a set of ***transactions and a chopping***, construct a graph  $Ch(T)$  such that

- the nodes of  $Ch(T)$  are the transaction pieces occurring in the chopping
- let  $p$  and  $q$  be 2 pieces from 2 different transactions. If  $p$  and  $q$  contain operations that are in conflict,  $Ch(T)$  contains an undirected edge between  $p$  and  $q$  labeled “C” (conflict)
- if  $p$  and  $p'$  are pieces from the same transaction  $Ch(T)$  contains an edge labeled “S” (sibling)



# Cycles in the Chopping Graph

- Isolation levels
- Transactional chopping
- Non-locking schedulers

Chopping graph may have ***cycles*** involving **s** or **c** edges.

An **sc** cycle is a cycle that contains at least one **c** and at least one **s** edge



# Example Chopping Graph

**Original transactions:**

$$t1 = r1(x)w1(x)r1(y)w1(y)$$

$$t2 = r2(x)w2(x)$$

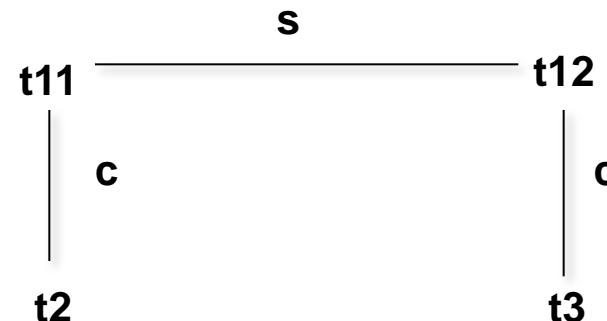
$$t3 = r3(y)w3(y)$$

Let  $t1$  be chopped into two pieces

$$t11 = r11(x)w11(x)$$

$$t12 = r12(y)w12(y)$$

**No sc cycle in chopping graph**



# Correct Choppings

-  Isolation levels
-  Transactional chopping
-  Non-locking schedulers

A ***chopping is correct*** if an execution according to the following rules is conflict equivalent to some serial history of the original transactions

1. When transaction pieces execute, they obey the precedence relations of the original transactions
2. Each piece will be executed according to a protocol ensuring conflict serializability and will commit

**Theorem:** A chopping is correct if the associated chopping graph does not contain an sc cycle



# Credit Card Scenario revisited

- Isolation levels
- Transactional chopping
- Non-locking schedulers

**Given:**

$$t1 = r1(A1)w1(A1)r1(B1)w1(B1)$$

$$t2 = r2(A3)w2(A3)r2(B1)w2(B1)$$

$$t3 = r3(A4)w3(A4)r3(B2)w3(B2)$$

$$t4 = r4(A2)$$

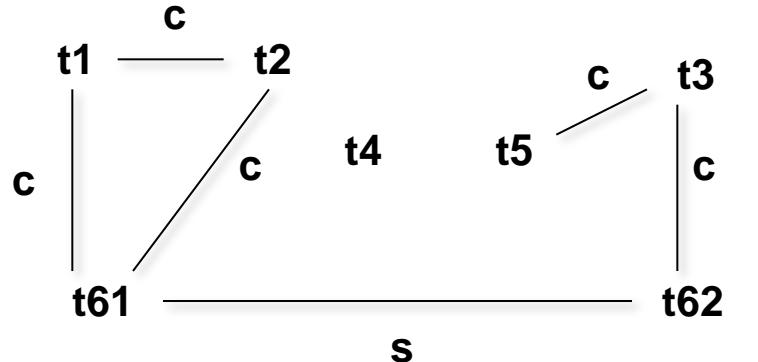
$$t5 = r5(A4)$$

$$t6 = r6(A1)r6(A2)r6(A3)r6(B1)r6(A4)r6(A5)r6(B2)$$

**Chop  $t6$  into**

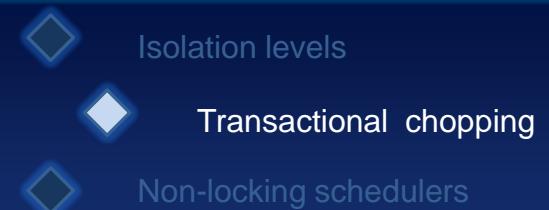
$$t61 = r61(A1)r61(A2)r61(A3)r61(B1)$$

$$t62 = r62(A4)r62(A5)r62(B2)$$



**No sc cycle, chopping is correct**

# Credit Card Scenario revisited



**Given:**

$$t1 = r1(A1)w1(A1)r1(B1)w1(B1)$$

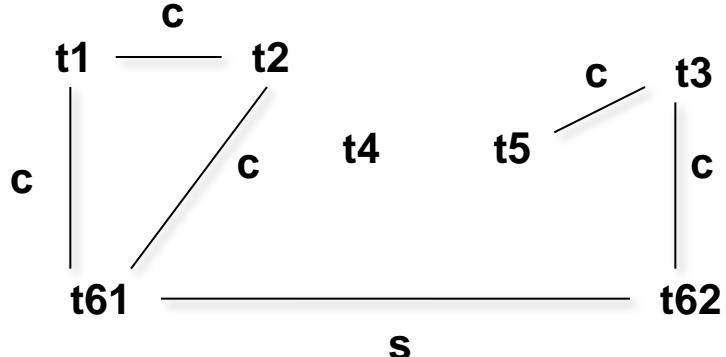
$$t2 = r2(A3)w2(A3)r2(B1)w2(B1)$$

$$t3 = r3(A4)w3(A4)r3(B2)w3(B2)$$

$$t4 = r4(A2)$$

$$t5 = r5(A4)$$

$$t6 = r6(A1)r6(A2)r6(A3)r6(B1)r6(A4)r6(A5)r6(B2)$$



No sc cycle, chopping is correct

**Chop t6 into**

$$t61 = r61(A1)r61(A2)r61(A3)r61(B1)$$

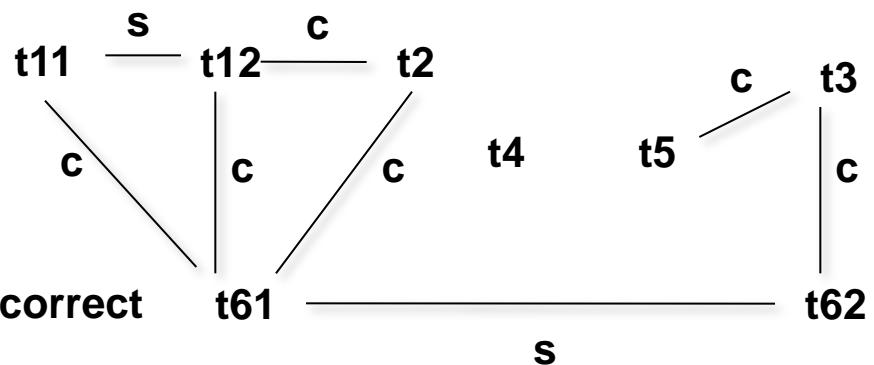
$$t62 = r62(A4)r62(A5)r62(B2)$$

**Chop t1 into**

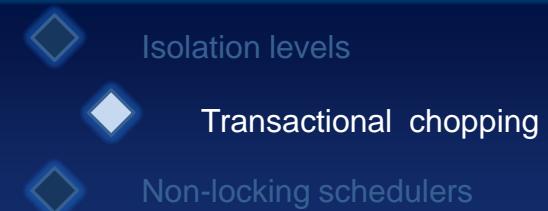
$$t11 = r11(A1)w11(A1)$$

$$t12 = r12(B1)w12(B1)$$

sc cycle  
chopping incorrect



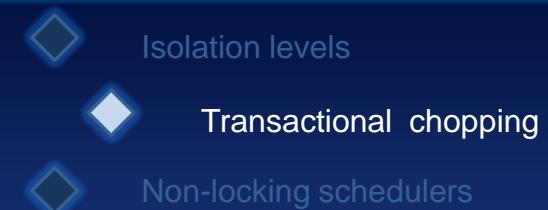
# SC Cycles in Chopping Graph



- Further chopping may introduce **SC cycles**
- Once an SC cycle exists in the chopping graph, no further chopping can remove the SC cycle
- **Algorithm** (sketch) for finding finest correct chopping
  - from given set of transactions  $\{t_1 \dots t_n\}$ , take a transaction  $i$  and chop it
  - substitute the chopping in the chopping graph for the whole set of transactions  $\{t_1, \dots, t_{i-1}, t_i, \dots, t_k, t_{i+1}, \dots, t_n\}$
  - if this graph does not contain an SC cycle, we found a private chopping for  $t_i$
  - if for each  $t$  in  $T$  a private chopping can be found, the optimum chopping is the union of all choppings



# Applicability of Chopping



- Chopping is done *at compile time* by inserting COMMIT WORK statements
- Even if statements are not in conflict at the conceptual level, conflicts may occur because underlying DBMS uses page level locking or some other shared structure
- ***Chopping algorithm*** applies for simple, straight line parameter-less SQL programs
- Programs with variables can be converted to straight line, parameter-less programs that cover the original program
- Applicability when program contains loops or other control structures (if-then-else) is an *open issue*

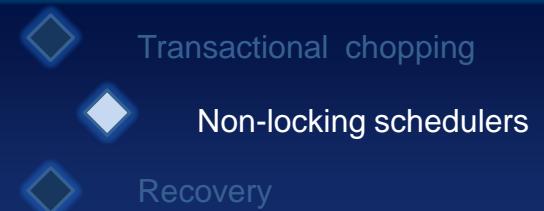


# Recommended reading

- For transaction chopping principle and example:
  - Weikum and Vossen, "Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery", Morgan Kaufmann Publishers, 2001
- For effects of transaction chopping and isolation levels on performance
  - Samuel Kounev, Alejandro Buchmann; Improving Data Access of J2EE Applications by Exploiting Asynchronous Processing and Caching Services, Proc. of the 28th International Conference on Very Large Data Bases - VLDB2002, August 2002



# Non-Locking Schedulers



- **Two big classes of non-locking schedulers**
  - time-stamp ordering schedulers
  - serialization graph testing schedulers
- **Optimistic schedulers**
  - defer validation phase to end of transaction



# Timestamp Ordering (TO)

- Transactional chopping
- Non-locking schedulers
- Recovery

- TM assigns each transaction a unique timestamp
- Operations of a transaction inherit its timestamp
- TO-rule:**  
if  $p_i[x]$  and  $q_k[x]$  are incompatible operations  $p_i [x]$  always executes before  $q_k [x]$  iff  $ts(T_i) < ts(T_k)$

## Theorem:

A history H produced by a TO-scheduler is serializable



# Proof of TO-Rule

-  Transactional chopping
-  Non-locking schedulers
-  Recovery

## Theorem:

A history H produced by a TO-scheduler is serializable

## Proof:

Let SG(H) be the serialization graph of H with edge  $T_i \rightarrow T_k$

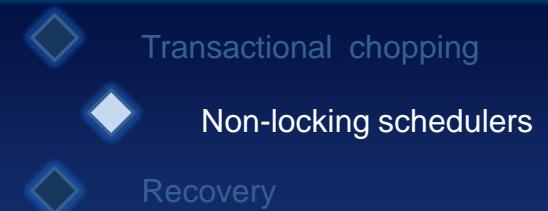
Two incompatible operations  $p_i[x]$  and  $q_k[x]$  must exist and they must be ordered  $p_i[x] < q_k[x]$

According to the TO-rule  $ts(T_i) < ts(T_k)$

If a cycle exists  $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$  and  $ts(T_1) < ts(T_1)$  which is a contradiction

Hence H is SR and a TO-scheduler produces SR histories





# Basic TO (BTO)

- **BTO schedulers** are aggressive, pass on ops. immediately
- Operations that arrive late according to TO-rule, i.e. arrive after an operation with a later timestamp, are rejected
- If an operation is rejected, the whole transaction must be aborted. It is restarted with a new timestamp
- A **BTO scheduler** must keep additional data structures to guarantee the proper execution order
  - for each object a counter with the ts of last read operation (*max-r-scheduled[x]*) and of the last write op. (*max-w-scheduled[x]*)
  - if for  $pi[x] \ ts(Ti) < max-w-scheduled[x]$ ,  $pi[x]$  is rejected
  - a counter of the operations in transit (*r-in-transit[x]* and *w-in-transit[x]*)
  - a *queue[x]* for each object containing ops. that could be executed according to TO-rule but are waiting for handshake



# Example of BTO

**Initial state:**  $\text{max-r-scheduled}[x]=0$ ,  $\text{max-w-scheduled}[x]=0$ ,  $\text{queue}[x]$  empty  
 $\text{TX-id} = \text{ts}$

1)  **$r_1[x]$  arrives**, is passed on and worked on by DM

$$\text{max-r-scheduled}[x]=1 \quad r\text{-in-transit}[x]=1$$

2)  **$w_2[x]$  arrives.** Although ts is ok, scheduler must wait for handshake since  $r\text{-in-transit}[x]=1$

$$\text{queue}[x] = w_2[x] \quad w\text{-in-transit}[x]=0$$

3)  **$r_4[x]$  arrives.** Although ts is ok, scheduler must wait for handshake ack<sub>2</sub>

$$\text{queue}[x] = w_2[x] \ r_4[x] \quad r\text{-in-transit}[x]=1 \quad w\text{-in-transit}[x]=0$$

4)  **$r_3[x]$  arrives.** Although ts is ok (no conflict in ops.)  $r_3[x]$  must wait

$$\text{queue}[x] = w_2[x] \ r_4[x]r_3[x] \quad r\text{-in-transit}[x]=1 \quad w\text{-in-transit}[x]=0$$





# Example of BTO

5) **ack( $r_1[x]$ ) comes back.** Scheduler sets  $r\text{-in-transit}[x]=0$ , takes  $w_2[x]$  from  $\text{queue}[x]$ , sets  $w\text{-in-transit}[x]=1$ ,  $\text{max-w-scheduled}[x]=2$

$\text{queue}[x] = r_4[x] \ r_3[x]$        $r\text{-in-transit}[x]=0$      $w\text{-in-transit}[x]=1$   
 $\text{max-w-scheduled}[x]=2$

6) **ack( $w_2[x]$ ) comes back.** Scheduler passes on  $r_4[x]$  and  $r_3[x]$  to DM

$\text{queue}[x] =$                            $r\text{-in-transit}[x]=2$                            $w\text{-in-transit}[x]=0$   
 $\text{max-w-scheduled}[x]=2$      $\text{max-r-scheduled}[x]=4$



# Strict TO (STO)

- **TO-rule** guarantees serializability but not RC

$$ts(T_1) = 1 \quad ts(T_2) = 2 \quad H = w_1[x] \ r_2[x] \ w_2[x] \ c_2$$

follows the TO-rule but is not RC (no  $c_1$  before  $c_2$ )

- **STO scheduler** resets  $w\text{-in-transit}[x]$  at end → no other operation can execute on  $x$
- All other operations wait in queue. Since conflicting ops. in queue are ordered by timestamp, no deadlock is possible
- Although  $w\text{-in-transit}[x]$  acts like a lock, STO produces other schedules than S2PL

$$H = r_2[x] \ w_3[x] \ c_3 \ w_1[y] \ c_1 \ r_2[y] \ w_2[z] \ c_2 \quad ts(T_1) < ts(T_2) < ts(T_3)$$

$H$  is equivalent to  $T_1 T_2 T_3$  and therefore serializable but can't be produced by S2PL because  $T_2$  would have to release lock on  $x$  before setting on  $y$

# Management of Timestamps



- Additional data structures are rather large, with small granules admin. information may be larger than DB
- **Need efficient *ts* management**
  - old *ts* are unlikely to be part of a conflict (assuming short transactions)
  - entries with old *ts* can be removed
  - if a transaction arrives with a *ts* that is older than the last threshold, the transaction is restarted
- If threshold values are used, TO-protocols must be amended accordingly (check first entry in ts-table to see if there is no change, if not, check against threshold)
- TO-protocols are susceptible to **differences in run-time**
- TO-protocols **easily distributed** (logical clocks)



# Optimistic Concurrency Control



- The execution of every transaction has 3 phases
  - ***validation*** phase where conflicts are determined
  - ***execution*** phase where operations of a transaction are executed
  - ***writing*** phase where update operations are written and committed
- The order of these phases determines whether a scheduler is pessimistic or optimistic
- **Pessimistic schedulers** assume that every operation is potentially conflicting and *validate before executing*
- **Optimistic schedulers** assume that conflicts are rare and execute an operation as it arrives and *validate later*
- If a conflict is detected during validation, a transaction involved in the conflict must be aborted



# Optimistic Schedulers



- **Optimistic scheduling protocols consist of**
  - *execution phase*
  - *validation phase*
  - *write/commit phase*
- Because they validate after execution, optimistic schedulers are also known as ***certifiers***
- **Serializability conditions:**
  - the writes of  $T_i$  may not collide with the reads of  $T_k$
  - the writes of  $T_i$  may not be overwritten by the writes of  $T_k$



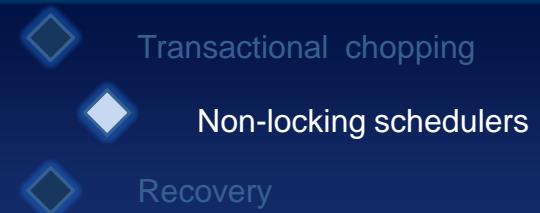
# Validation in Optimistic Concurrency Control



- **Validation can occur in 2 distinct ways:**
  - *Backwards validation:* validating transaction checks conflicts against committed transactions it overlapped with. If a conflict is detected, validating transaction must abort.
  - *Forward validation:* validating transaction checks conflicts with active transactions. Either the validating or another conflicting transaction may be aborted
- The validation mechanism is based on *comparing read and write sets*. The conflict criteria can be either those of lock-based, TO or SGT schedulers



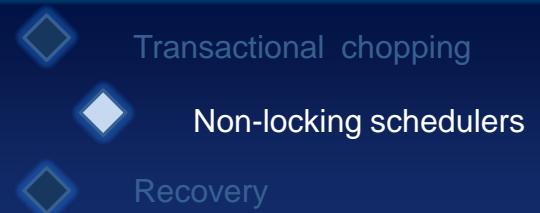
# Forward Validation



- *Optimistic schedulers* pass an operation immediately to DM
- When a  $c_i$  arrives the certifier invokes the validation and checks if an operation  $p_i[x]$  conflicts with any other operation  $q_k[x]$ . In this case  $k$  may be any active transaction
- An *optimistic scheduler* needs the following data structures:
  - list of all active transactions
  - the read-set of every active transaction ( $r\text{-scheduled}[T_i]$ )
  - the write-set of every active transaction ( $w\text{-scheduled}[T_i]$ )
- **Conflicts are detected by forming the intersections**
  - $(r\text{-scheduled}[T_i]) \cap (w\text{-scheduled}[T_k])$
  - $(w\text{-scheduled}[T_i]) \cap (r\text{-scheduled}[T_k])$
  - $(w\text{-scheduled}[T_i]) \cap (w\text{-scheduled}[T_k])$



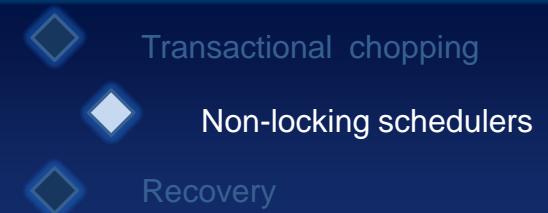
# Backward Validation



- In ***backward validation*** the set of transactions against which the validating transaction validates is the set of time-wise overlapping committed transactions
- In case of conflict, only the validating transaction can be aborted



# Serializability of Optimistic Schedules



- A 2PL scheduler produces the ***same schedules*** as an optimistic scheduler with locking-like conflict resolution
- An insertion into the read-set or write-set is ***equivalent*** to a lock request
- ***Validation tests*** for conflicts among entries in the read-sets and write-sets, which is equivalent to testing for lock conflicts



# Performance of Optimistic Schedulers

Transactional chopping

Non-locking schedulers

Recovery

- **Optimistic schedulers** let transactions continue in spite of potential conflicts →
  - optimistic schedulers perform roughly equal to locking 2PL schedulers when few conflicts occur
  - when conflicts increase the useless work of aborted transactions favors locking 2PL schedulers
  - optimistic TO schedulers let transactions continue when they would be aborted by a BTO scheduler, therefore they are less efficient
- Optimistic scheduling has *not* proven to be sufficiently better than conventional (pessimistic) locking approaches
- Optimistic ***multiversion*** concurrency control is used when there is a large portion of reading transactions



# Multiversion Concurrency Control

- So far we have assumed a single version of a data object in the database
  - Write operations overwrite a data item
- If we keep old versions in the database we have multiple consistent states but of differing actuality (freshness)
- Since storage space is finite, only a limited number of versions could be held in practice
- Concurrency control theory must be extended for multiple versions



# Implications of MV Concurrency Control

- The theory of serializability can be extended to MV
  - MV view serializability is NP complete
  - MV conflict serializability produces a subset of MVVS
- MVTO (multi version timestamp ordering)
  - Processes operations in FIFO order
  - Converts operations on data into operations on data versions
  - Each version carries the timestamp of the transaction that created it



# MV2PL Protocol

- Similar to MVTO, a scheduler can be defined that uses locks and a strong (strict) two-phase discipline
  - First simplification uses infinite number of copies
  - Special case uses exactly 2 versions



# Snapshot Isolation

- *Snapshot Isolation* is a special case of MVCC in which the version function always maps to the last version written before the transaction started and the write sets of pairs of transactions are disjoint
- *In Snapshot Isolation  $T_i$  executes against a snapshot of the database:*
  - Comprising the versions of each data item  $T_i$  accesses, and that were committed before  $T_i$  started
  - And its own changes
- Reads are never blocked – useful for read mostly scenarios
  - Leverage SSD read performance (later)
- Implemented in Oracle, PostgreSQL, SQL Server



# Snapshot Isolation [1] - Basics

- $\text{BOT}_i = \text{timestamp}(\text{Begin}_{-T_i})$ 
    - $R_i[X] \quad W_i[Y] \quad W_i[X] \quad R_i[Y]$
  - $EOT_i = \text{Commit} \rightarrow \text{timestamp}(\text{End}_{-T_i})$
- 
- $R_i[X]$  – read the last version of X committed before  $T_i$  started,
    - NO READ locks
    - $< \text{timestamp}(\text{Begin}_{-T_i})$
    - If  $T_i$  already modified a data item  $\rightarrow$  sees its own version, e.g.  $R_i[Y]$
  - $W_i[X]$  – Concurrent transactions, modifying the same data item cannot commit
    - Check First-Committer-Wins-Rule or First-Updater-Wins-Rule
  - Commit:  $T_i$  commits unless  $T_i$  concurrent with  $T_j$  and  $\text{Writeset}(T_i)$  overlaps  $\text{Writeset}(T_k)$

[1] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. 1995. A critique of ANSI SQL isolation levels. In *Proc. The ACM SIGMOD'95 (San Jose, California, United States, May 22 - 25, 1995)*



# Snapshot Isolation – First-Committer-Wins

- First-Committer-Wins-Rule or First-Updater-Wins-Rule → prevent Lost Update
- First-Committer-Wins-Rule
  - Executed while committing
    - If for the duration of Tx another transaction has written and committed a common item → abort



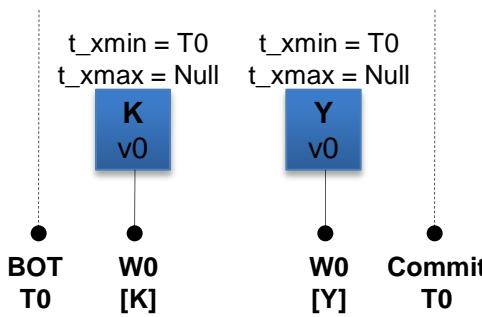
# Snapshot Isolation – First-Updater-Wins

- First-Updater-Wins-Rule
  - Long X-locks for Writes (updates, deletes)
  - Before Wi [K] , Ti requests an X-Lock. If granted check:
    - If K already updated by concurrent Tj → abort | otherwise commit
  - If lock NOT granted (concurrent Tj holds lock). → Wait until Tj terminates
    - If Tj commits → abort
    - If Tj aborts → acquire lock, check if concurrent transaction has updated K. If NO→commit



# Snapshot Isolation in PostgreSQL [2,3] – 1

- Tuples are unit of versioning
- TID<sub>i</sub> – unique Transaction ID of transaction T<sub>i</sub>, assigned by the scheduler
  - Equivalent to timestamp( Begin\_T<sub>i</sub> )
- Each Version V<sub>j</sub> of a tuple K is annotated with two TIDs (HeapTupleFields structure):
  - t\_xmin – TID of the transaction that created V<sub>j</sub>
  - t\_xmax – TID of the transaction that invalidated V<sub>j</sub> by updating K (and creating a new version) → t\_xmax=TID<sub>y</sub>
    - If no succeeding version exists → t\_max = NULL
  - forward link to new versions – all versions are stored as a linked list



[2] H. Korth, A. Silberschatz. *Database System Concepts*. McGraw-Hill Publ. Comp., 2001  
[3] S. Wu, B. Kemme. *Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation*. Proc. of the IEEE ICDE, Tokyo, Japan, 2005

# Snapshot Isolation in PostgreSQL – 2

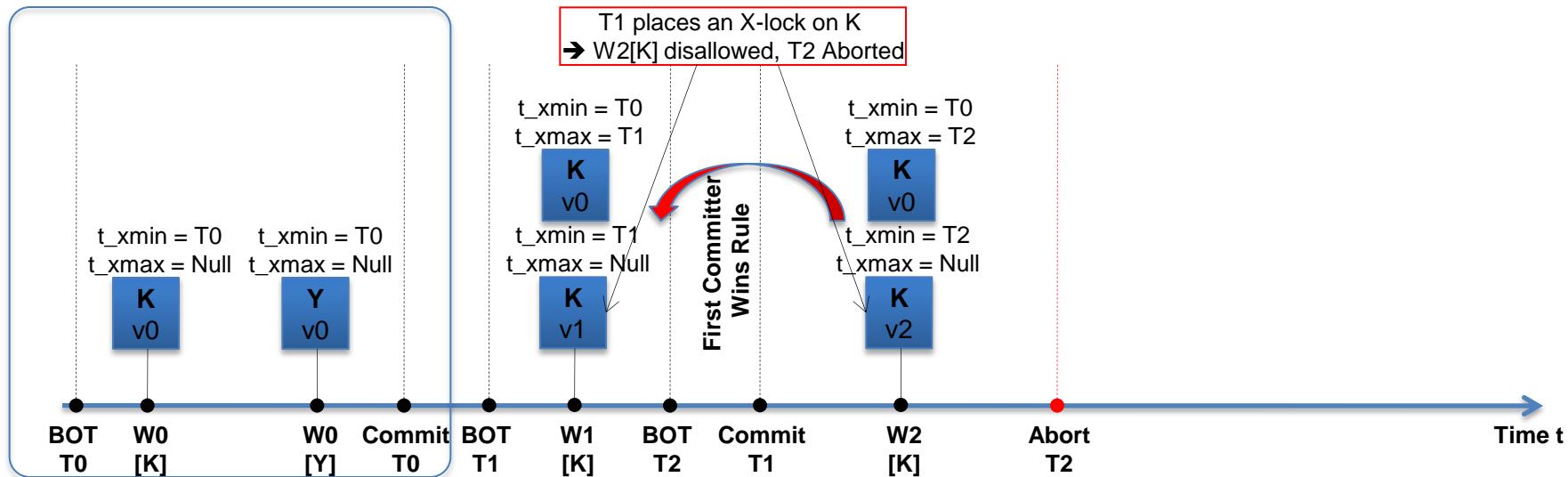
- How to determine the “latest” version of a tuple for a TX to use?
  - **Tuple Visibility Concept** [2,4] - for Read and Write
- PostgreSQL maintains per running transaction  $T_i$ :
  - SnapshotData – structure
    - $xmax$  – the TID of the next transaction (at time  $T_i$  started) → serves as threshold
    - $xmin$  – transactions whose updates are visible to  $T_i$
    - $xip$  – TID list of all active transactions
  - PG\_CLOG (previously pg\_log) – metadata about status of transactions
    - Check transaction status (aborted, committed, in progress) – fast
- Version  $V_a$  of Tuple K ( $K.V_a$ ) is visible to transaction  $T_i$  iff:
  - $K.V_a$  was created by  $T_j$  that committed before  $T_i$  made its version
  - $K.V_a$  was updated by  $T_j$  that either aborted or started after  $T_i$  made its version or was in progress when  $T_i$  made its version

[4] Tom Lane. Transaction Processing in PostgreSQL.  
<http://www.postgresql.org/files/developer/transactions.pdf>



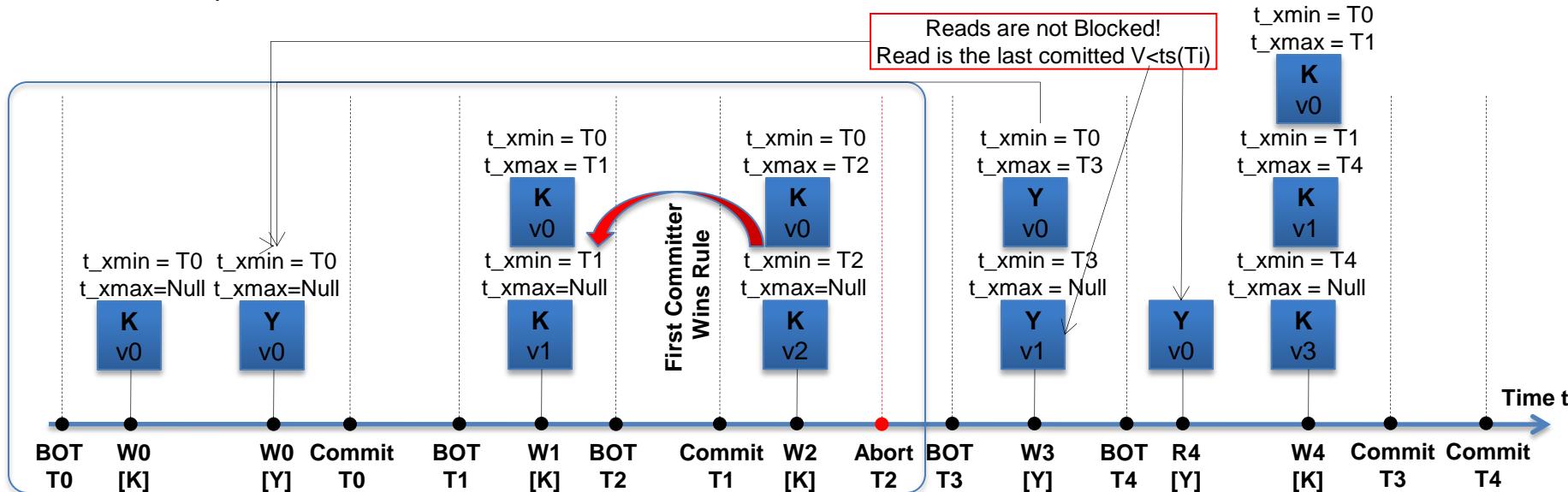
# Snapshot Isolation in PostgreSQL – 3

- First Updater Wins Rule – implemented using long exclusive locks
  - Before  $T_i$  performs  $W_i[K]$  – performs version check: → If it fails → Abort
  - Otherwise  $T_i$  requests a long X-lock (released at the end of transaction  $T_i$ )
  - Updates → 2 writes: (i) update  $t_{xmax}$  and (ii) create the new version
  - If X-Lock Granted to  $T_j$  (no updates)  $T_i$  waits for  $T_j$  to terminate
    - ... (the rest is according to the general description of the rule)



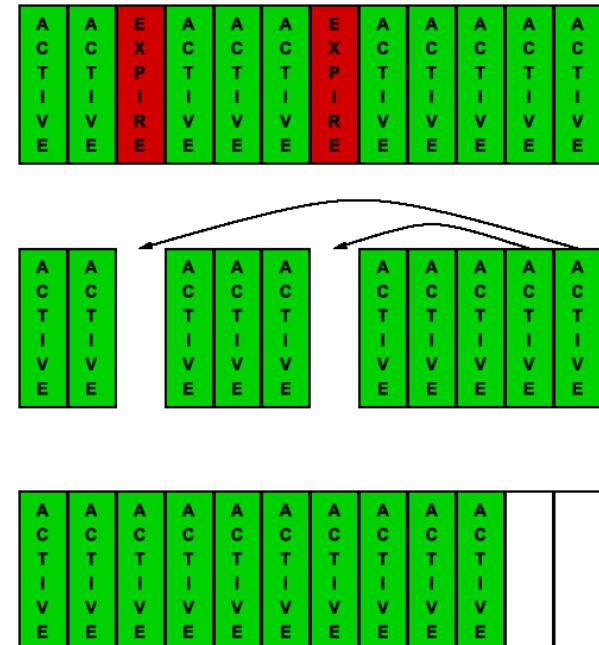
# Snapshot Isolation in PostgreSQL – 4

- Lock-free Reads
- If  $T_i$  performs  $W_i [K] \rightarrow R_i [K]$  reads its own version
- Else: Read  $K_{Vj}$ , created by  $T_j$ :  
 $\text{timestamp}(\text{commit\_}T_j) < \text{timestamp}(\text{Begin}_T)$ 
  - $t_{xmin} = TID_j$
  - $t_{xmax} | (t_{xmax} = \text{Null}) \text{ or } (t_{xmax} \rightarrow \text{Aborted TID}) \text{ or } (t_{xmax} = \text{concurrent TID})$



# Snapshot Isolation in PostgreSQL – 5

- Version Management in PostgreSQL
- Problem Old versions (invisible to Tx) remain as dead bodies
- Reclaimed by VACUUM – if not VISIBLE to other Tx
- Simple VACUUM
  - versions marked as invalid
  - space marked as free
  - NO coalescing
- Exhaustive VACUUM
  - Coalescing → Moves tuples across blocks
  - High IO Costs
  - Exclusive Table Locks



# Snapshot Isolation and SSDs

- SI → Lock-free reads – maximum use of SSD read performance
- (Random) Reads – determine the read-set and write-set
- Random Writes
  - Create new versions
  - Change  $t_{xmax}$  of existing versions
- Low Latency
- Fast Random Reads
- Slow Random Writes
- Asymmetric behavior
  - Random Read I/Ops faster than Random Write
  - Sequential Read faster than Sequential Write
- Better random throughput for small block sizes (4k better than 16k)



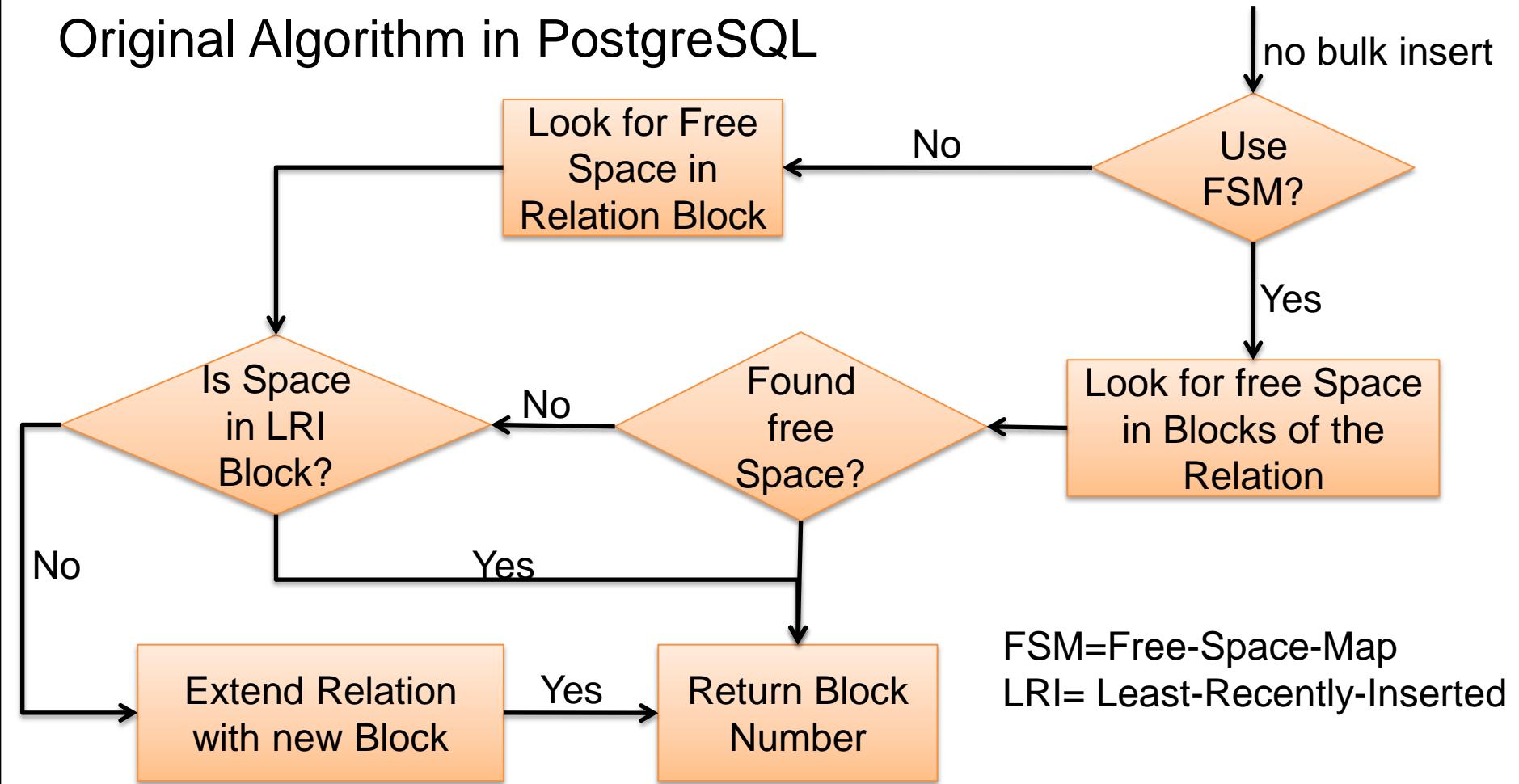
# Concept: SI-CV

- Idea: Convert random writes to sequential writes for Snapshot Isolation
- SI-CV → Version is changed at Insert/ Update (changed t\_max)
  - Insert/ Update through HeapIO (heapam.c and hio.c)
- Exploit the principle of multiple versions of one item → save new Versions into one block
- Assumption: A transaction (TX) inserts values (versions of values)  $V_i$  into the DB
  - Instead of assigning  $V_i$  addresses on the disc, make a reservation of multiple adjacent addresses
  - On the next insert/update from TX, the adjacent address is chosen and gets assigned to the TX
  - TX that inserts several Versions of Items will receive addresses that linger on the same block(s) on the SSD



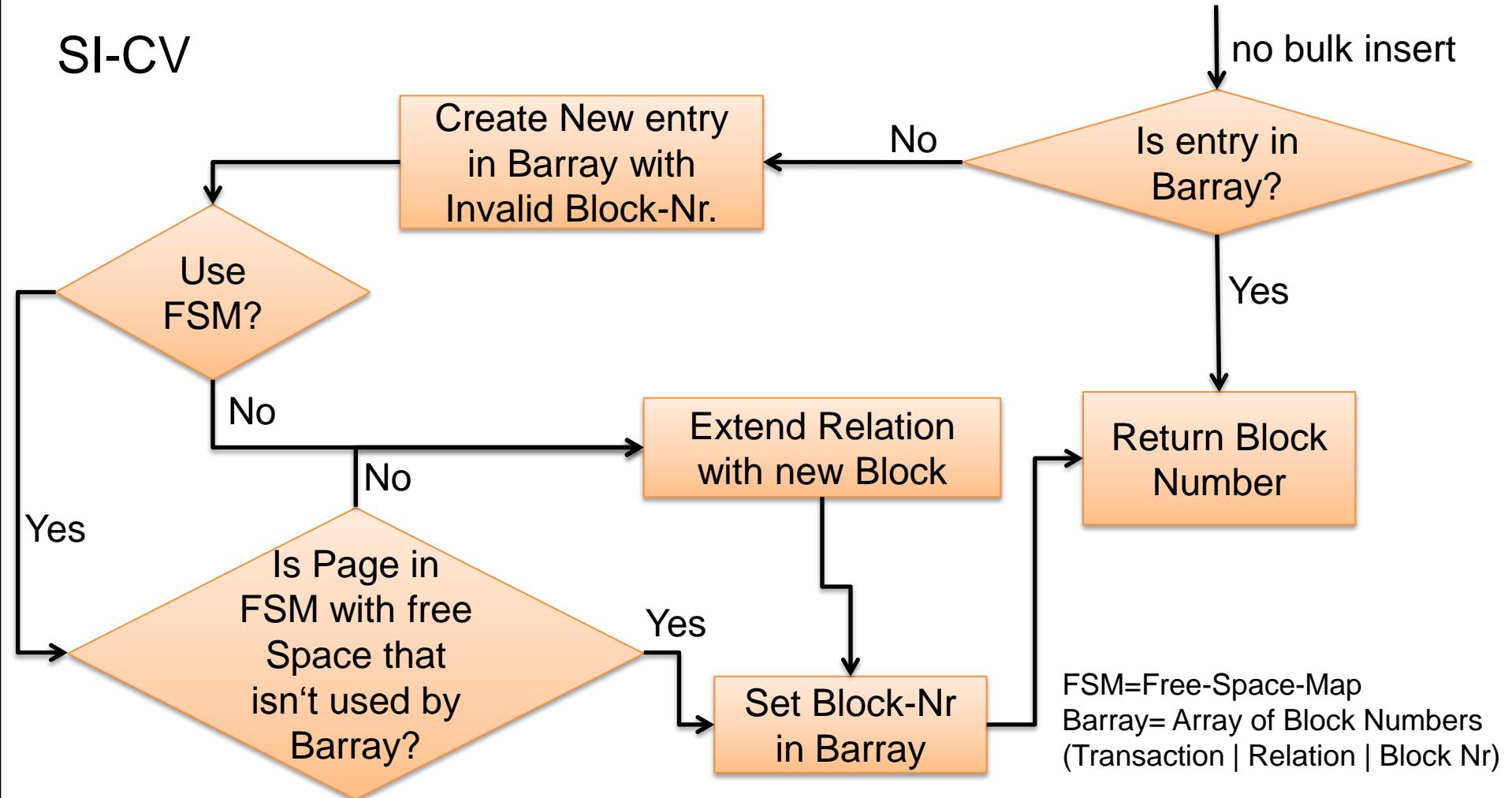
# Snapshot Isolation with Co-Located Versions

## Original Algorithm in PostgreSQL



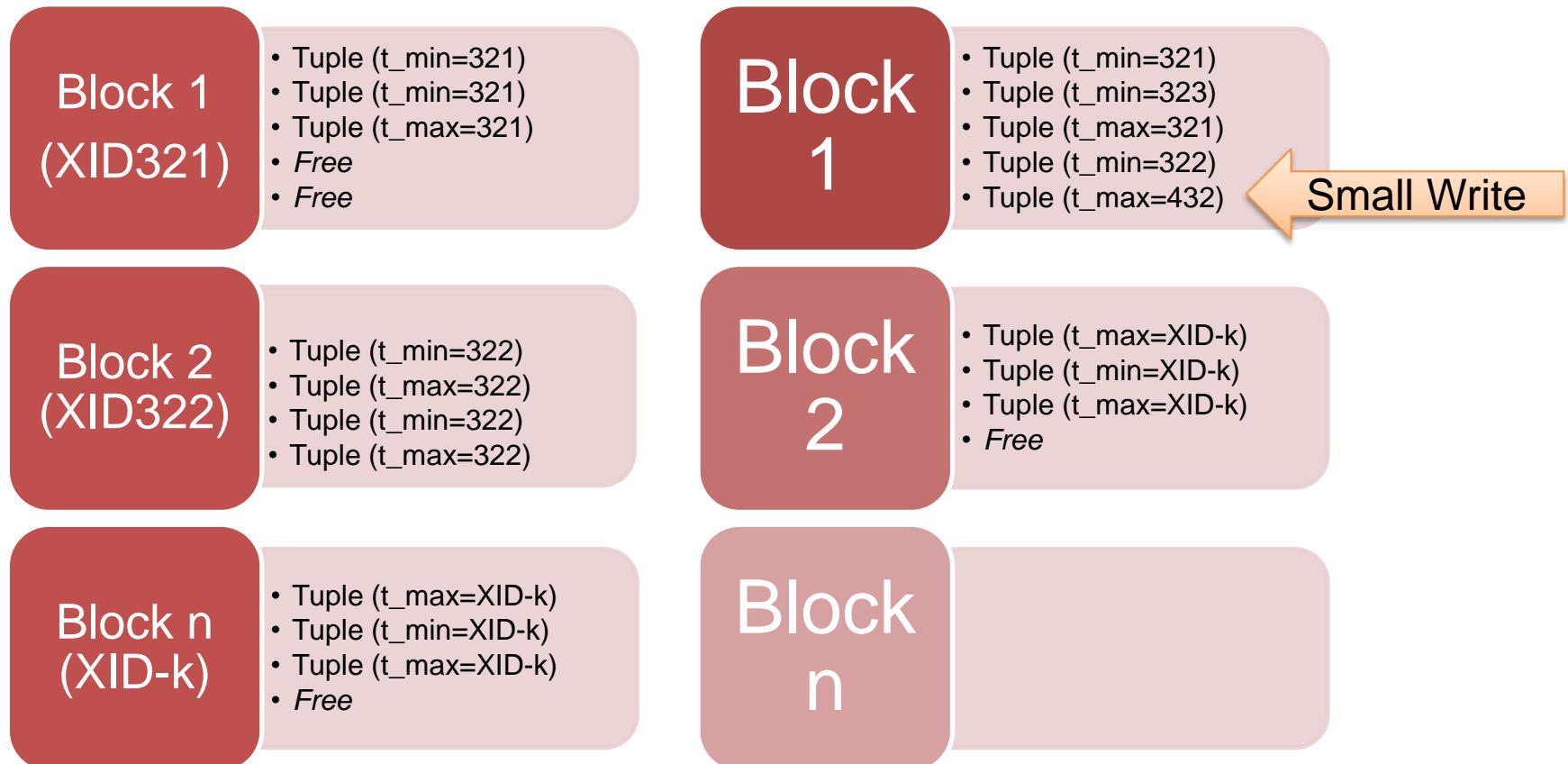
# Snapshot Isolation with Co-Located Versions

SI-CV

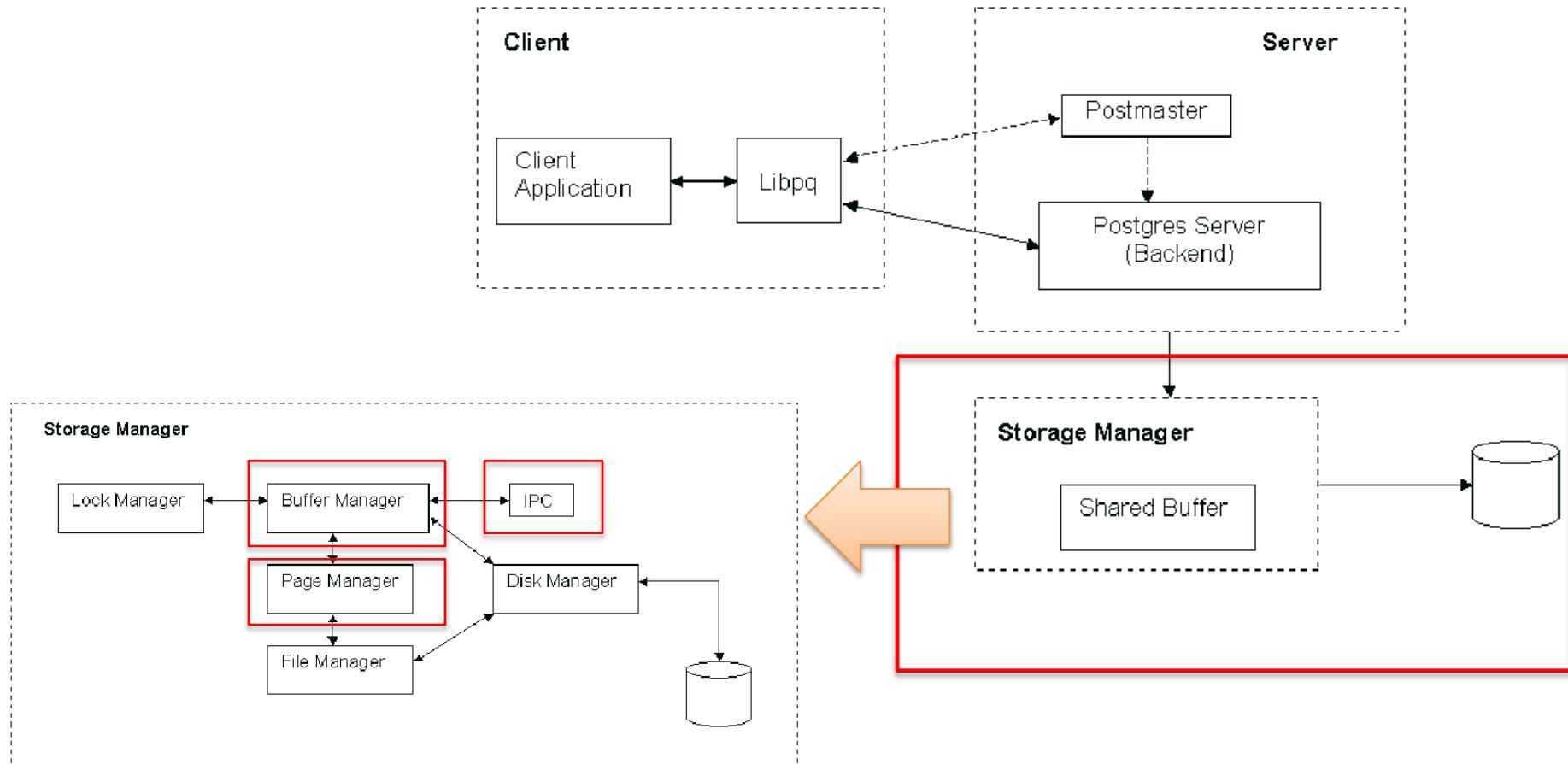


# Snapshot Isolation with Co-Located Versions

## Co Located Versions



# Snapshot Isolation with Co-Located Versions



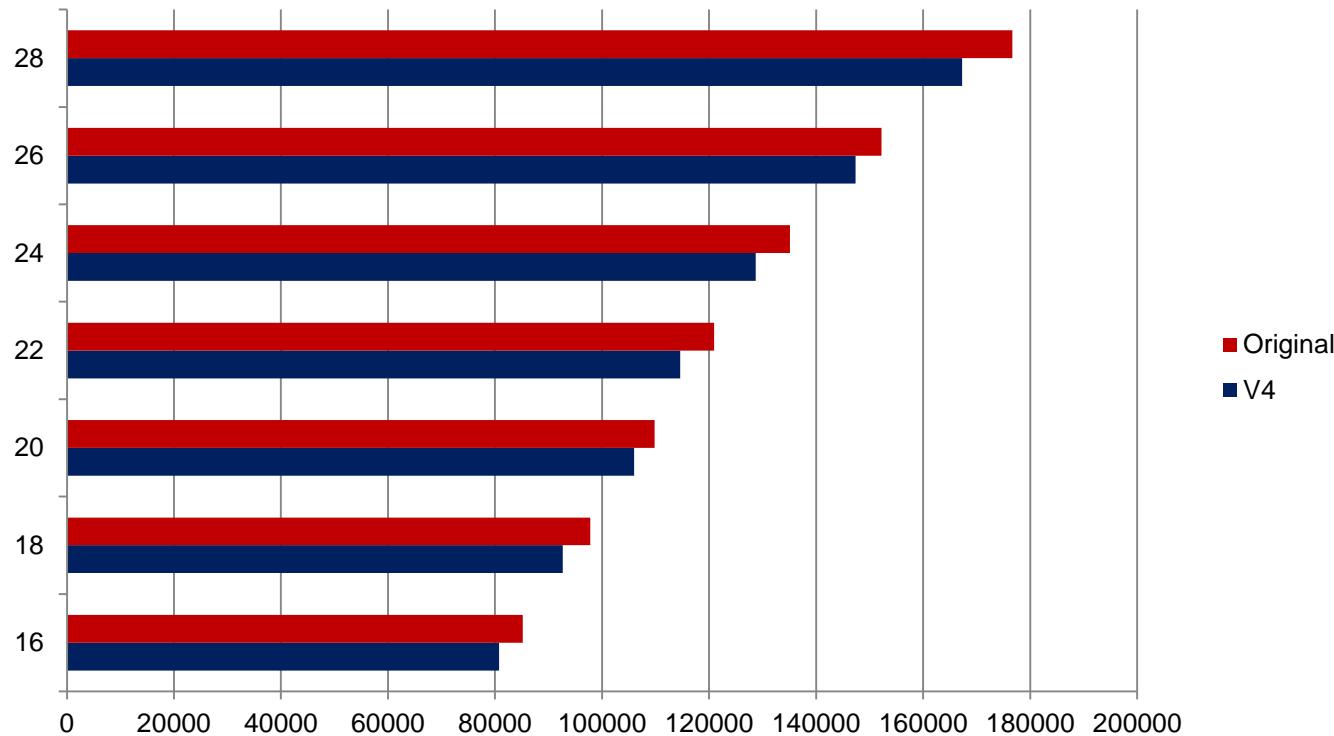
# Micro – Benchmarks

- Settings:
  - Inserts
- To generate as many Versions as possible we use a mixture of Updates and Reads:
  1. Creation of Table with 1 000 000 entries; Set UpdateTX=X
  - 2.a Read of whole Table (generate Version i)  
→ No Commit, keep cursor & read table once
  - 2.b Update Table (generates Version i+1)  
→ Wait for Update to finish
  - 2.c UpdateTX--;
  3. while (UpdateTX>0) goto 2.a
  4. Disconnect and Commit all open/ uncommitted Transactions



# Benchmarks – SSD

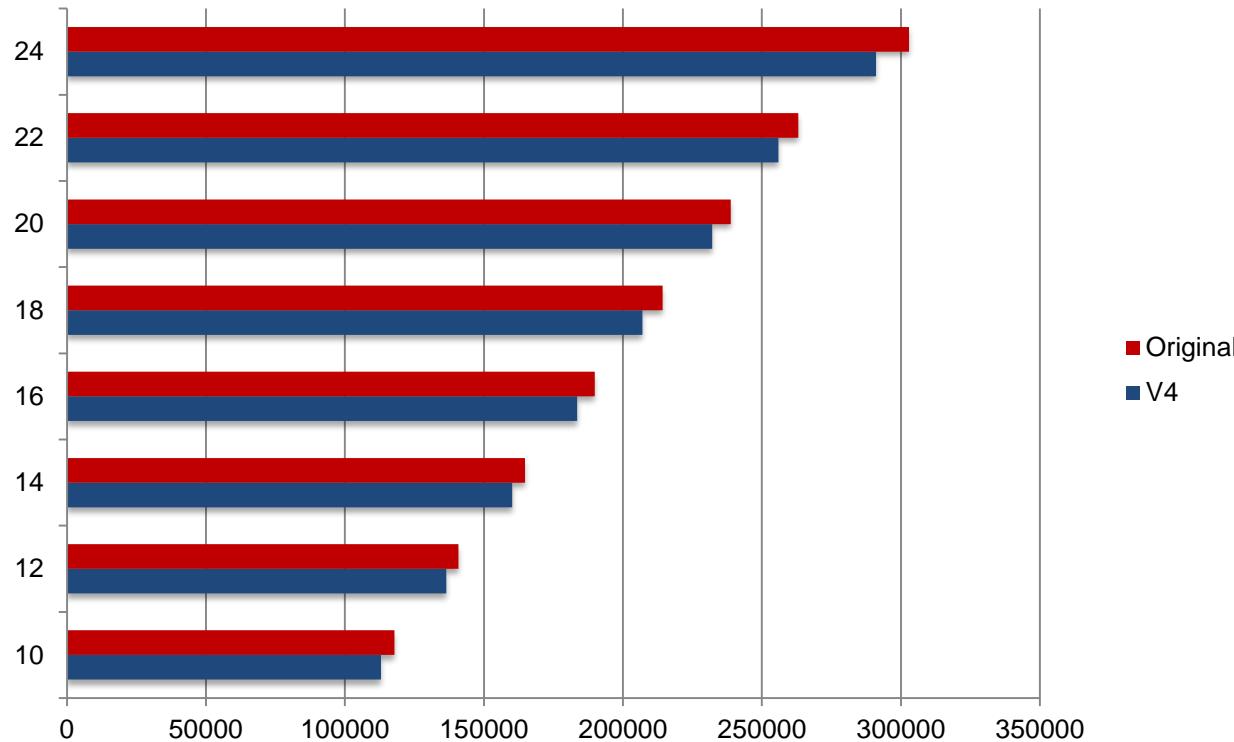
SSD: Update & Read 1000000 Entries



- SI-CV „V4“ is >5% faster than original PostgreSQL
- Setting: Multiple transactions which concurrently read & update the DB

# Benchmarks – HDD

HDD: Update & Read 1 000 000 Entries



- SI-CV implementation increases Speed on HDD
- On HDD - equal to or faster than the original PostgreSQL SI

# Recovery

- Out of the 4 ACID properties, **recovery** deals with
  - *atomicity* (all or none of the actions of a transaction are executed)
  - *durability* (changes are permanent in the physical DB)
- **Recovery** guarantees that a consistent state is reestablished in case of faults
  - *logical faults* - transaction is stopped based on internal problems (e.g. input problems, arithmetic problems, missing data). TX is not restarted
  - *system faults* - TX is stopped due to system fault (e.g. deadlock). TX is automatically restarted
  - *system crash* - system fails (e.g. DBMS failure, OS failure, HW failure), content of main memory corrupted or lost
  - *disk failure* - write fault or head crash, one or more blocks unreadable

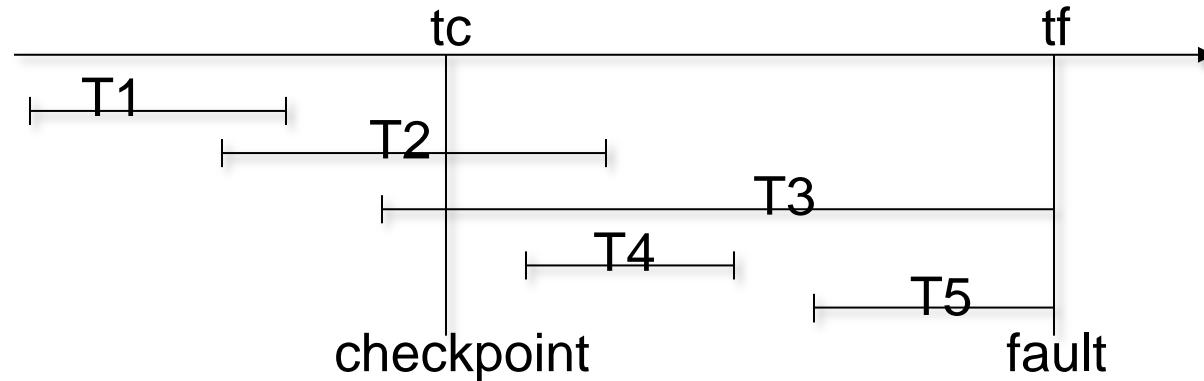
# Recovery Manager vs. Buffer Manager



- Recovery and Buffer Managers are **not independent**
- BM may *replace* uncommitted pages which overwrite old data
- BM may *keep* committed but frequently used pages in buffer
- Depending on what policy is enforced the state of the DB may vary after a crash
  - *RM allows BM* to flush uncommitted pages to disk → STEAL
  - *RM forces BM* to keep modified pages in buffer until transaction commits → NO STEAL (simplifies UNDO)
  - *RM forces BM* to propagate (flush) pages with committed changes to disk → FORCE (avoids partial REDO)
  - *RM may allow BM* to keep modified pages in buffer → NO FORCE
- **Recovery algorithms must consider RM/BM interaction**



# Recovery Classes (Gray)



- **R1 - partial UNDO** (after transaction fault)
  - isolated rollback of one or more transactions because of TX fault
- **R2 - partial REDO** (after system failure)
  - redo of all completed transactions whose data was still in buffer
- **R3 - global UNDO** (after system failure)
  - removal of pages of uncommitted TX that were pushed to disk
- **R4 - global REDO** (after media failure)
  - copy archival copy, redo committed TXs since date of archival copy



# Logs



- Recovery is based on redundancy
- **Redundant information kept in form of logs**
  - *temporary log files*
  - *archive log files*

	state	transition
logical	-	operators and arguments
physical	before/after images	XOR difference

- Write ahead logging assumes that log entries are written to stable storage before overwriting the corresponding data



# Physical Logging



- Physical storage units are also ***logging units*** (e.g. pages)
- For each page that is loaded for update into buffer a before-image is kept when update in place is used (only one per page per TX, oldest image is enough)
- Because write operation may fail, after-image is written to log before data is written to disk
- ***R1 (partial UNDO)*** and ***R3 (global UNDO)*** are realized with before images
- ***R2 (partial REDO)*** and ***R4 (global REDO)*** require after images



# Physical Logging with Pages and Transitions



- In ***Physical Logging with Pages and Transitions*** only the difference between two pages is stored
- Difference is obtained through  $\text{XOR}$
- $\text{Difference} \text{ XOR } \text{old page} = \text{new page}$  (and viceversa)
- Differences are easily compressed



# Logical Logging

- **Logical logging** is performed at record level and assumes a consistent initial state
- **Generic structure of a log record**
  - record-id
  - transaction-id
  - ptr-previous
  - DB-id
  - relation-id
  - record-address
  - #fields-updated
    - field-id, old-value, new-value

# Simple Recovery Algorithm w. UNDO/REDO

- We assume a Steal/No-force policy
  - Active transactions must be undone (dirty pages may have been written)
  - Committed transactions must be redone (committed changes may not have been made to the physical DB)
- Simple 2-pass algorithm:
  - Get active transaction list
  - From end of log work backwards and undo all changes made by active transactions (apply before-images)
  - From appropriate point in log (e.g. last checkpoint) redo changes of committed transactions (apply after-images)



# Overview of Aries



- **Aries** developed for DB2,
- **Aries** works with steal, no-force policy
- Recovery Manager is invoked after crash, 3 phases:
  - **analysis**: identifies dirty pages in buffer pool (i.e. changes that have not been written to disk) and active transactions
  - **redo**: repeats all actions starting from an appropriate point in log, restores the DB to the state it had at the time of the crash
  - **undo**: undoes the actions of all the transactions that did not commit, so that the DB reflects only the actions of committed TXs



# Recovery Example



LSN	LOG
10	Update: T1 writes P5
20	Update: T2 writes P3
30	T2 commit
40	T2 end
50	Update: T3 writes P1
60	Update: T3 writes P3
X	Crash, Restart

- At restart, **analysis** identifies
  - $T1, T3$  as transactions active at crash, must be undone
  - $T2$  committed, all actions must be written to disk
  - $P1, P3, P5$  potentially dirty pages
- **Redo phase**
  - all updates are reapplied in order (including those of  $T1$  and  $T3$ )
- **Undo phase**
  - uncommitted updates are undone in reverse order (update  $T3$  to  $P3$ , update  $T3$  to  $P1$ , update  $T1$  to  $P5$ )



# Principles Behind Aries



- **Write-ahead Logging:** every change to a DB-object is first written to the log; the log record must be written to stable storage before the change to the DB is written to disk
- **Repeating history during Redo:** upon restart DB is brought into the exact state it had at time of crash. Only then uncommitted transactions are aborted during Undo.
- **Logging changes during Undo:** changes made during Undo are logged to avoid applying them multiple times in case of failure during restart
- **Restart algorithm must be idempotent** (if applied multiple times it produces same result as applied once)



# Discussion of Aries' principles



- Repeating history during Redo distinguishes Aries from other recovery algorithms, gives *more flexibility*
- Since DB is brought to the exact state as before crash, changes of any granularity can be *undone in third phase*
- Aries works with page level locking and with record level locking



# The Aries Log



- Every log record has a unique identifier called a ***Log Sequence Number (LSN)***
- **LSN** is monotonically increasing → determines order of log records
- Every page in the DB contains the LSN of the *most recent log record* that describes a change to that page (page LSN)
- The most recent portion of the log is the ***log tail***. It is kept in memory and is flushed periodically to disk or when a page is to be written and its log entry has not been made permanent



# Recording in the Aries Log



- **Updating a page:** after a page is modified, an update log record is appended to the log tail. PageLSN is set to LSN of update log record

LSN	PrevLSN	TransID	type	pageID	length	offset	Before image	After- image
-----	---------	---------	------	--------	--------	--------	-----------------	-----------------

Common to all log records

Specific for update log records

- **Commit:** a commit-type log record containing the transaction ID is force-written to the log tail. Force-write causes log tail to be flushed to disk. Transaction is removed from active transaction table



# Recording in the Aries Log

- **Abort:** when a transaction is aborted, an abort type record is appended to the log tail and Undo is initiated for that transaction
- **End:** since some clean-up is needed after commit or abort (e.g. removing transaction from transaction table), an end-type log record is inserted when cleanup is completed.
- **Undoing an Update:** when transaction is rolled back (transaction abort or rollback after crash) its updates are undone and for each undone update a compensation log record (CLR) is written

# Compensation Log Records



- **CLRs** are needed to record actions taken to undo an update
- **CLRs** are written to record an action that is *irreversible* (once an abort is decided, it must be executed), therefore CLRs don't contain information to reverse the undo action
- **CLRs** point to the next undo to be carried out (prevLSN of the undone update)

Dirty P. recLSN	prevLSN	txID	type	pageID	length	offset	before	after
P500		T1000	U	P500	3	21	ABC	DEF
P600		T2000	U	P600	3	41	HIJ	KLM
P505		T2000	U	P500	3	30	GDE	QRS
TX-ID lastLSN		T1000	U	P505	3	21	TUV	XYZ
T1000		T1000	C	P505	3	21		TUV
T2000								

# Transaction and Dirty Page Tables



- **Transaction Table:**
  - contains one entry for each active transaction
  - entries contain transaction ID, status (in progress, committed, aborted), lastLSN (points to last log entry for TX)
  - if status is aborted or committed, entry is removed after clean-up
- **Dirty Page Table:**
  - contains an entry for each modified page in buffer pool that has not been written to disk
  - recLSN is the LSN for first log record that caused page to become dirty (earliest log record that might have to be redone for this page during restart)



# Write Ahead Logging (WAL)



- **WAL** states that any change must be logged first to stable storage before it is applied to the DB
- **Force all log records** upto and including that corresponding to the pageLSN of the page being written to disk (not part of generic WAL but used when totally ordered log is needed, e.g. ARIES for repeating history)
- A commit forces all the log records of a transaction's changes plus the commit record to stable storage
- Cost of forcing log records to stable storage is much smaller than cost of forcing the modified pages to disk



# Checkpointing



- **Checkpoints** are like snapshots of the database
- **Goal** is to produce a stable state of the database
- **Trade-off:** checkpoints cost during normal operation but reduce amount of work to be done at restart
- **Three ways of achieving stable state:**
  - transaction consistent checkpointing
  - action consistent checkpointing
  - fuzzy checkpointing

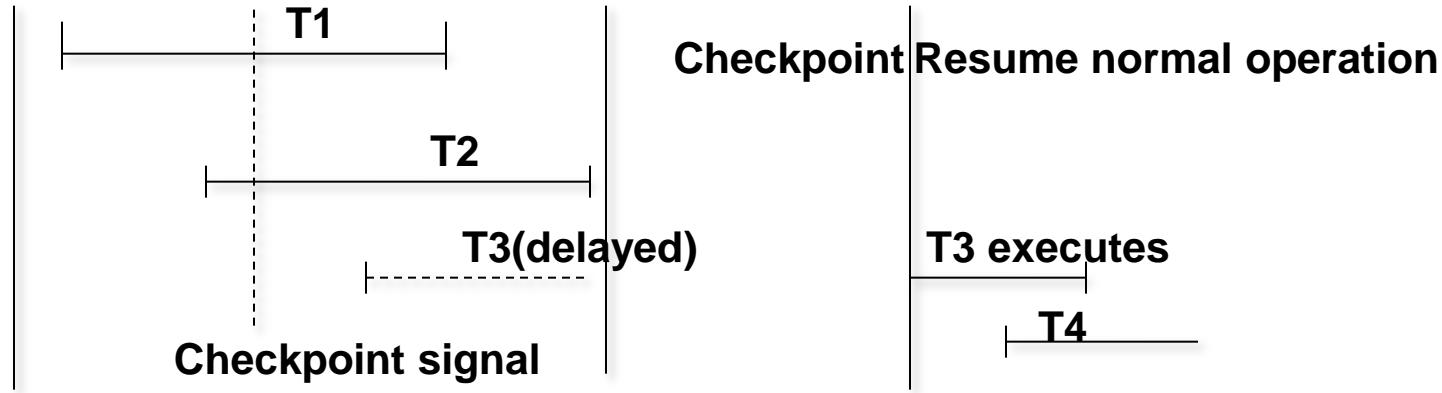


# Transaction Consistent Checkpointing

Non-locking schedulers

Recovery

Replication



- Simplest and most *inefficient* method to achieve stability is to let all transactions commit before taking a snapshot
- When checkpoint signal is given, running transactions finish, newly arriving transactions are delayed
- When all running transactions have committed, *checkpoint is made* (buffer is flushed to disk, log is marked)
- *After checkpoint delayed transactions are started*



# Discussion of Transaction Consistent Checkpointing



- ***Very simple but inefficient*** since individual transactions are delayed until
  - a) all running transactions finish and
  - b) checkpoint is completed
- Restart works backward to last checkpoint mark (UNDO) and forward from there (REDO)



# Cache Consistent (Action Consistent) Checkpointing

Non-locking schedulers

Recovery

Replication

- **Action consistent checkpointing** doesn't require transactions to finish, only individual actions
- No new transactions are permitted to start
- Active transactions are *blocked* (no new actions are started)
- **Checkpointing consists of 2 parts:**
  - mark the log, commit list, abort list to indicate what has been reflected in physical DB, current log buffer forced to disk
  - all dirty pages are forced to disk
  - materialized DB now contains everything that was changed in buffer
  - special log entry (CKPT,List) is written to disk to mark the checkpoint as complete. List contains all active transactions at time of checkpoint



# Discussion of Cache Consistent Checkpointing



- No transaction has to wait for other transactions to commit → ***shorter delays***
- **Restart** must do *less work*
  - all committed transactions are guaranteed to be in materialized DB up to the checkpoint marker → no REDO for all transactions that committed before checkpoint
  - all aborted transactions were already made invalid up to the checkpoint, therefore less UNDO



# Fuzzy Checkpointing

- A third way of *approaching* stability is fuzzy checkpointing
- **Fuzzy checkpointing** offers an optimization that minimizes interruptions (transactions don't wait for writes)
- Buffer is not flushed and *DB-operation may proceed during checkpointing*
  - stop new updates, commits and aborts
  - scan cache, produce list of changed pages
  - produce list of active transactions and ptr. to most recent log entry
  - write log record (incl. active transactions and pointers) to stable log
  - accept new updates, commits and aborts
  - in background force dirty pages to disk

# Fuzzy Checkpointing (cont.)



- New checkpoint record only written when all dirty pages on disk
- Once new checkpoint is written, previous state is guaranteed
- This kind of fuzzy checkpointing guarantees ***penultimate*** state



# Fuzzy Checkpointing ARIES-Style



- **Begin-checkpoint record** is written to log.
- An **end-checkpoint record** is constructed that contains the content of a) *Transaction table* and b) *Dirty page table*.
- Write end-checkpoint record to log
- Write *master record* to a separate, known location of stable storage. *Master record* contains LSN of Begin-checkpoint
- While end-checkpoint record is being constructed, DBMS continues executing transactions → transaction table and dirty page table are accurate as of writing of begin-checkpoint
- Background process flushes dirty pages from end-checkpoint record to disk



# Discussion of Fuzzy Checkpointing



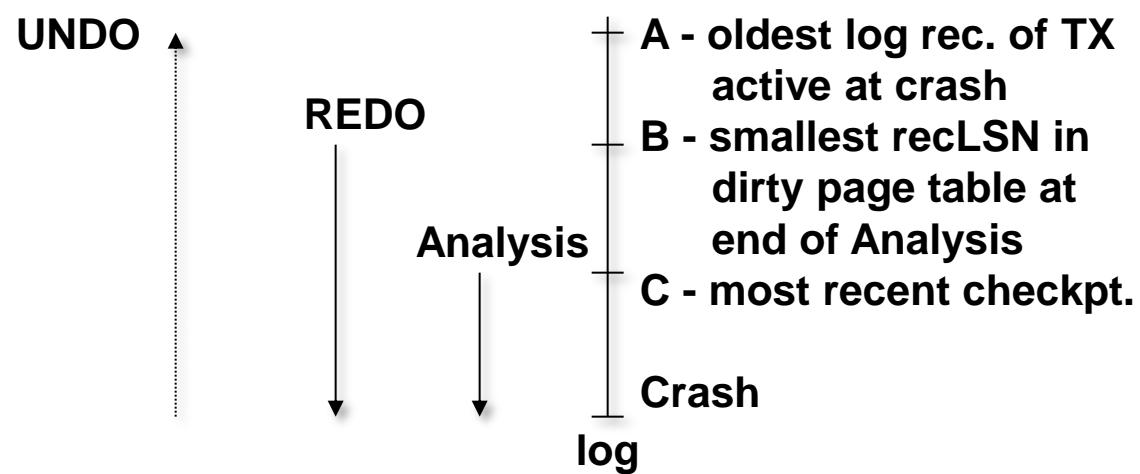
- **Inexpensive**, since it doesn't require to quiesce the system (transaction consistent) or writing out buffer during checkpointing (cache consistent)
- Depending on implementation of fuzzy checkpointing, in general consistent penultimate state can be guaranteed (generic fuzzy checkpointing)
- Effectiveness limited by earliest recLSN of pages in Dirty Pages Table (ARIES fuzzy checkpointing) since during restart REDO must start at this recLSN
- ARIES fuzzy checkpointing depends on its version of WAL which requires writing all log entries to disk when page is flushed (up to recLSN for page to be flushed)



# Recovering from a System Crash



- After a system crash the **Recovery Manager** must execute the *restart algorithm*
- Simplest restart algorithms are **two-pass algorithms**: first pass undoes aborted transactions, second pass redoes committed transactions
- Will analyze ARIES restart algorithm (3 phases)
  - *analysis phase*
  - *redo phase*
  - *undo phase*



# Analysis Phase



- ***Analysis phase begins*** at most recent begin-checkpoint record (which was written to master record in known loc.)
- **Identifies** transactions that were active at time of crash
  - initialize dirty page table and list of active transactions from most recent end-checkpoint record
  - if an end log record is found, remove that transaction from active list
  - if a log record is found for a transaction not on active list, insert that transaction into active list
  - if a log record for an active transaction is found, update the lastLSN field to that of this record, set status to Undo unless it is a commit log record in which case status is set to C



# Analysis Phase (cont.)



- ***Analysis phase determines*** (a conservative superset of) pages in buffer pool that were dirty at time of crash
  - if a redoable log record is found check if affected page is in dirty page table
  - if affected page is not in dirty page table, insert it and set recLSN to LSN of this log record (this LSN identifies the *oldest* change affecting the page - note direction of processing!)
  - all dirty pages are guaranteed to be in dirty page table at end of analysis phase, but some pages that were later written to disk might also be included in dirty page table



# Redo Phase



- The ***redo phase*** reestablishes the state at time of crash
- All actions are redone, incl. those of aborted transactions
- ***Redo*** starts at log record with smallest recLSN among those in dirty pages (oldest update that may not have been written to disk)
- **Scan forward until end of log.** For each record encountered:
  - for each update log record or CLR check whether action must be redone



# Redo Phase (cont.)



- **action must be redone unless**
  - affected page is not in dirty page table (see note)
  - page is in dirty page table but recLSN > LSN being checked (update was propagated to disk)
  - pageLSN of page that must be retrieved to check condition is greater than or equal to LSN of log record being checked (either this update or a later update caused page to be written to disk)
- **NOTE:** According to analysis phase described here, there is no possibility that affected pages could not be in dirty page table. However, if file-level operations are permitted (e.g. dropping of files or extents) this might happen.



# Redo Phase (cont.)



- since recLSN indicates first update to this page which may not have been propagated, if  $\text{recLSN} > \text{LSN}$  no action is required
- third condition requires fetching the page to check whether the write was propagated or a later update to the page was written.
- **If logged action must be redone:**
  - apply logged action
  - set pageLSN on the page to LSN of redone record



# Undo Phase



- **Undo** scans *log backwards* and aborts all transactions that were active at time of crash (all marked with status=U in active transaction table after analysis phase)
- Transactions to be undone are called **losers**
- Losers must be undone in *reverse order* starting with their *lastLSN*
  - construct ToUndo set inserting the lastLSN for each loser
  - select the highest LSN in ToUndo
  - process record, insert prevLSN into ToUndo set (unless it is null)
  - repeat until toUndo is empty



# Processing the Undo



- **Processing a log record** in the Undo phase *requires*:
  - if it is an update record, undo action and write a CLR, insert prevLSN in update record into ToUndo set
  - if it is a CLR and the undoNextLSN is not null, add undoNextLSN to ToUndo set.
  - If it is a CLR and the undoNextLSN is null, discard CLR and insert an end record
- Once the ToUndo set is *empty*, the Undo phase is *complete*, and so is restart
- Resume normal operation
- Aborting a single transaction is the same with only one value in ToUndo

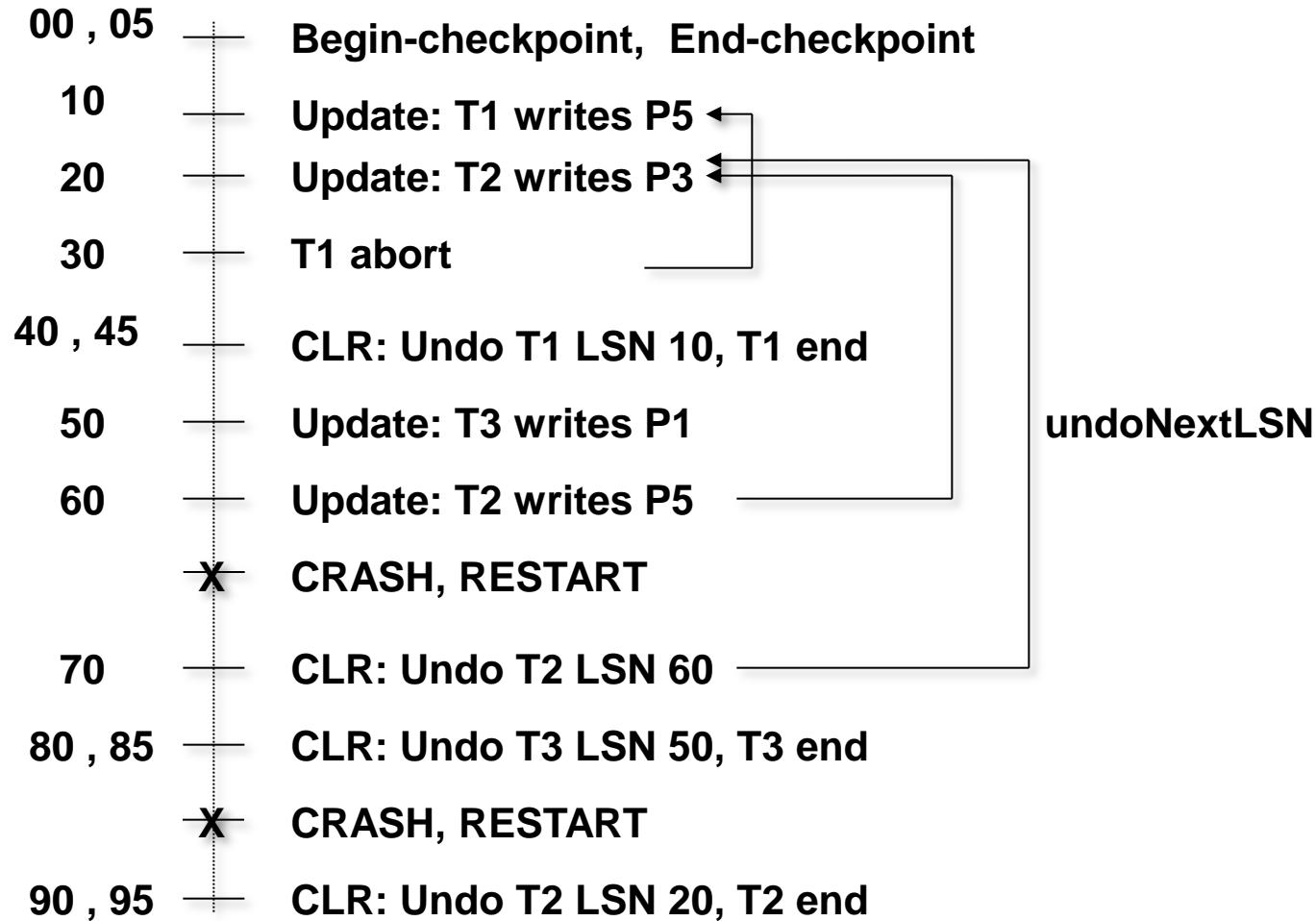


# Example of Undo with Repeated Crashes

Non-locking schedulers

Recovery

Replication



# Media Recovery



- Based on **periodical copies of database**
- Creating a copy is done *similar* to fuzzy checkpointing
- Only **committed transactions** are kept, uncommitted transactions and their changes are eliminated from backup (equivalent to transaction consistent checkpoint)
- **If single page is corrupted**
  - use backup copy of that page and reapply changes from committed transactions in log and undo changes of uncommitted transactions



# Replication (in Oracle)

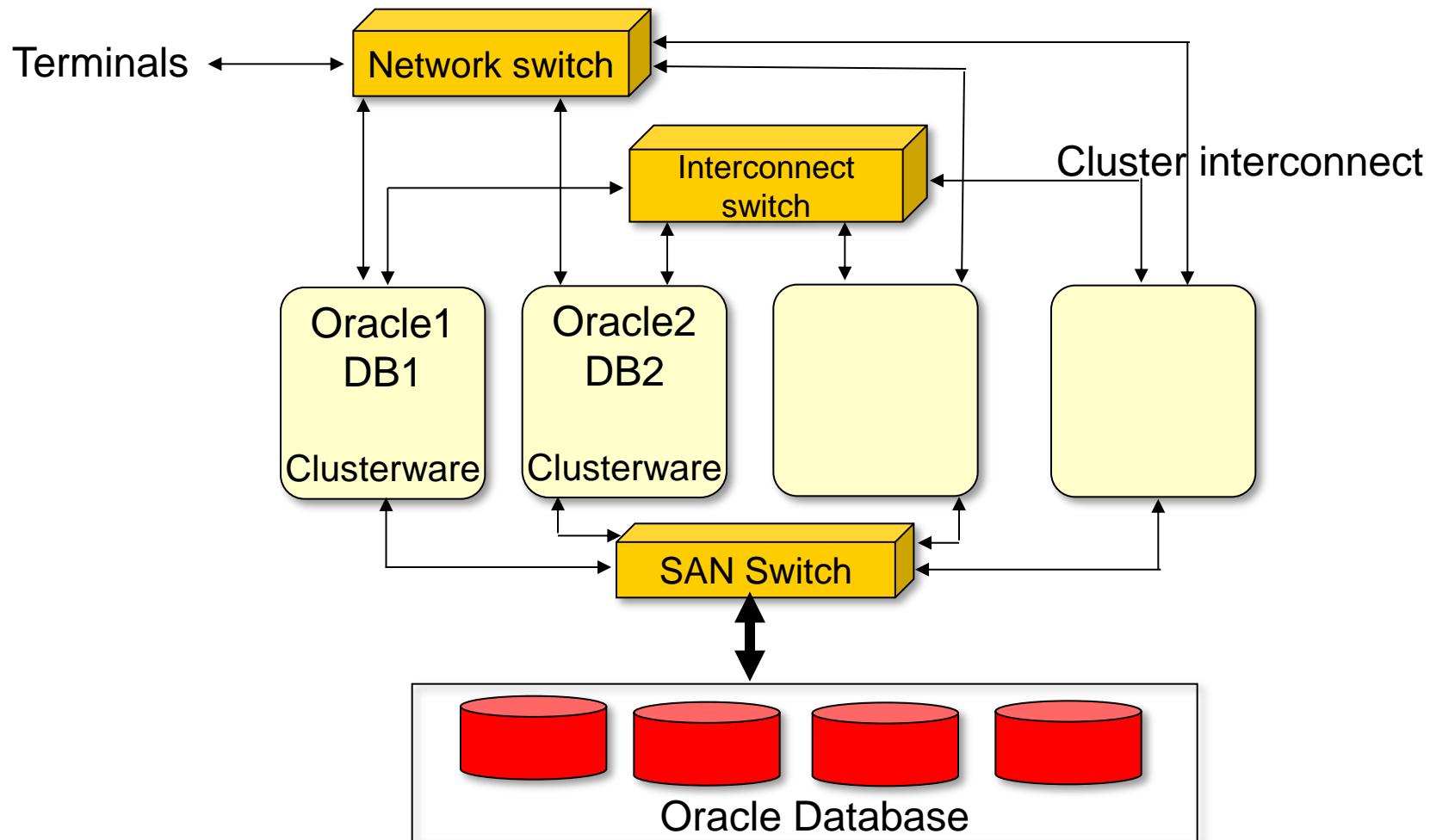


- **Three different approaches to replication in Oracle**
  - *RAC (Real Application Cluster)*
  - *Data Guard*
  - *Streams*
- **RAC is a clustered database solution**
  - requires 2 or more nodes configured under a clustered OS
  - cluster management software
    - maintains cluster coherence
    - manages common components (e.g. disk subsystem)
  - **examples:**
    - HP Tru64, Sun Cluster, Veritas Cluster Manager, Oracle Clusterware



# RAC

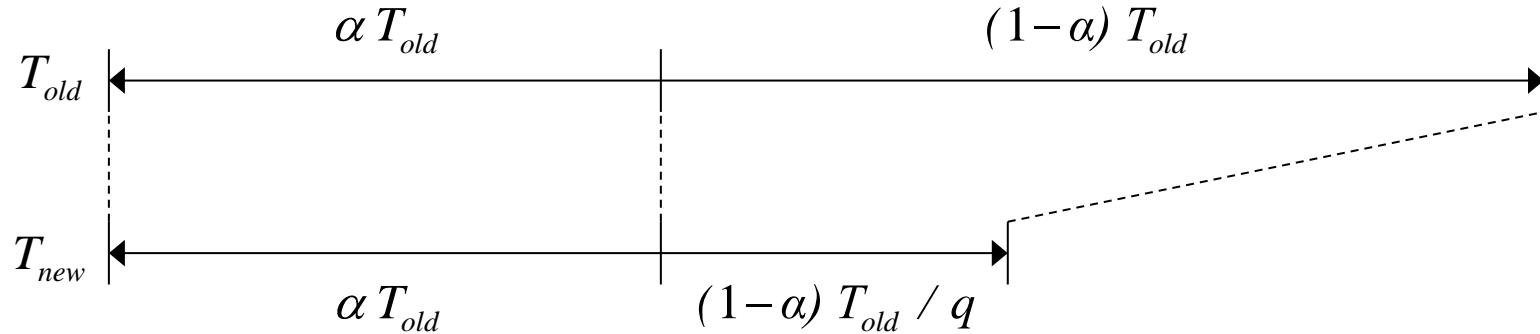
- Non-locking schedulers
- Recovery
- Replication



- Many ***instances*** of Oracle *running* on many nodes
- Many instances *sharing* a single physical database
- All instances have *common* data and control files
- Each instance has *individual log files* and undo segments
- All instances are able to *execute simultaneously* transactions against the single physical DB
- Instances participating in the clustered configuration communicate via cluster interconnect using cache fusion

- **RAC offers scalability**
  - Multiple instances can be added
  - Each instance has its own hardware resources
  - Typical scale-out
- **RAC offers high availability**
  - Built in feature since database can be accessed by any of the DBMS instances, failure of one node doesn't cause database failure
- **RAC uses shared storage to host the database structures**

# Amdahl's Law



Fraction  $(1 - \alpha)$  of the execution time is boosted by a factor of  $q$

$$S = \frac{T_{old}}{T_{new}} = \frac{T_{old}}{\alpha T_{old} + (1-\alpha)T_{old}/q} = \frac{1}{1/q + \alpha(1-1/q)}$$

$$\lim_{q \rightarrow \infty} S = \lim_{q \rightarrow \infty} \frac{1}{1/q + \alpha(1-1/q)} = \frac{1}{\alpha}$$

Fixing the amount of work leads to an upper bound on how much the overall performance can be improved by enhancing a single system component or the benefit attainable from parallelization

# Data Guard



- ***Data Guard*** provides a *complete* standby database
- More a **Disaster Recovery solution**
  - peer host system with independent database
  - complete and identical replica of the primary database
  - can be used immediately when the primary database fails
- Reads redo log to extract the database changes
- Redo log entries extracted by the log transport service



# Oracle Streams



- **Stream based replication can provide**
  - alternative data sources
  - subsets of primary database or complete replica
  - data can be sent to multiple destinations
  - data can be transformed via apply handlers
  - auditing routines can be applied and tracked
  - heterogeneous Streams can export data to non-Oracle databases
  - data from non-Oracle databases can be imported via User Application Enqueue mechanisms
  - streams can be used in RAC environments



# Oracle STREAMS and Advanced Queueing



- **Advanced Queueing** allows enqueueing of messages and propagation of messages to subscribing queues
- AQ sends notifications when a message arrives in a queue
- **AQ provides**
  - multi-consumer queues
  - publish/subscribe mechanisms
  - content based routing
  - transformations
  - gateways to other systems
  - persistence even after messages are dequeued



# Streams and Data Replication



- **Streams** are used mostly for data replication
- **Oracle Streams** keeps track of database changes and captures them as events
- DB changes (DML or DDL) are wrapped and formatted into *Logical Change Records (LCR)*
- LCRs are propagated and applied at the target DB
- Stream based replication allows *asynchronous* replication
  - LCRs extracted from redo log files (after these are written by background log writer process, does NOT access redo log buffer)
  - does not delay normal transaction processing of source DB
  - capture process can also read the archived redo logs
  - capture process is defined by rules that dictate whether a log entry must be extracted or discarded



# Flash and Databases – Recommended Reading

- Lee, Sang-Won, Moon, Bongki; “Design of Flash-Based DBMS: An In-Page Logging Approach, SIGMOD 2007, Beijing, June 2007.
- Lee, Sang-Won, Moon, Bangki, Park, Chanik, Kim, Jae-Myung, Kim, Sang-Woo; A Case for Flash Memory SSD in Enterprise Database Applications, SIGMOD 2008, Vancouver, June 2008.
- Graefe, Goetz; The Five-Minute Rule 20 Years Later, ACM Queue, July/August 2008.
- Chien, Andrew, Karamacheti, Vijay; Moore’s Law: The First Ending and a New Beginning, IEEE Computer, December 2013
- Li, Yan, Quader, Khandker; NAND Flash Memory: Challenges and Opportunities, IEEE Computer, August 2013
- Huffman, Amber, Juenemann, Dale; The Nonvolatile Memory Transformation of Client Storage, , IEEE Computer, August 2013

# Properties of Flash Memory

- Two types of flash memory
  - NAND: cheap, blocked access
  - NOR: expensive, direct addressing
- NAND flash has become quite affordable, used in many mobile devices and in solid state disks (SSD)
- Form factor equivalent to rotating magnetic disks, also common interface available
- No rotational/mechanical parts → low power consumption, low latency, shock resistant
- Limiting factors: no selective overwrite → must erase and rewrite block, approx. 100 000 write cycles max.



# Properties and cost of flash (Figures for 2007 used by Graefe)

	RAM	Flash	SATA disk
Price and capacity	\$3 for 8x64 Mbit	\$999 for 32 GB	\$80 for 250 GB
2014 prices		\$0.80 per GB	\$0.10 per GB
Transfer bandwidth		66 MB/s API	300 MB/s API
Access latency		0.1 msec	12 ms
Active power		1 W	10 W
Idle power		0.1 W	8 W
Sleep power		0.1 W	1 W

Note: prices are plummeting and capacity is expected to double every year till 2012  
What happened since?



# NAND-Flash and Moore's Law

- Moore's Law states that the ***economically producible*** components will double in density approximately every 18 months
- The economic aspect is often ignored
- Moore's Law held for NAND-Flash for the past 10 years
- But...
  - Increasing capacity requires multiple layers
  - Multiple layers increase interference and more sophisticated charge management
  - Reliability suffer → write capacity has peaked, reliable reads have stagnated
- But...
  - Flash meets most applications' needs and new persistent memory techniques are being developed



# Flash vs. Disk Performance

Storage	Hard disk	Flash SSD
Avg. latency	8.33 msec	0.2 msec read 0.4 msec write
Sustained transfer rate	110 MB/sec	56 MB/sec read 32 MB/sec write

Disk: Seagate Barracuda 7200.10 ST3250310AS

SSD: Samsung MCAQE32G8APP-0XA drive with K9WAG08U1A  
16 Gbits SLC NAND chips

# Where and how can flash be used?

- Different use of NAND Flash in DBMS and in file system
  - Flash as part of buffer pool (typical for file systems)
  - Flash as part of fast persistent storage (typical of DBMS)
- Possibilities for using flash in DBMS
  - Indexes, table spaces → small random writes (slow) but very fast random read
  - Logging space, temporary data space, rollback segments → large sequential writes (fast)
  - SIAS (Snapshot Isolation w. Append Storage)



# Impact of flash on logging

- At commit time the tail of the log must be forced to persistent storage
- $T_{\text{resp}} = T_{\text{cpu}} + T_{\text{read}} + T_{\text{write}} + T_{\text{commit}}$
- Fast CPUs and large data caches and asynchronous write (no force option) make  $T_{\text{commit}}$  more critical
- Group commits alleviate the logging bottleneck and increase throughput but delay individual transactions
- Since log records are always appended and written to a separate device, this is ideal for flash SSD
  - Log writing latency for rotating disks is  $\frac{1}{2}$  revolution = 4.17 msec
  - Log writing latency for SSD is 0.4 msec



# Impact on query processing

- Simple SQL transactions accessing and writing 1 record and committing
- 2 identical systems, commercial DBMS, one system with disk and the other with SSD
- Entire table in memory to eliminate effects of  $T_{cpu}$ ,  $T_{read}$  and  $T_{write}$

# concurrent Transactions	hard disk		flash SSD	
	TPS	%CPU	TPS	%CPU
4	178	2.5	2222	28
8	358	4.5	4050	47
16	711	8.5	6274	77
32	1403	20	5953	84
64	2737	38	5701	84



# Impact on TPC-B benchmark

- TPC-B benchmark is obsolete but stresses different subsystems and causes many small transactions with significant forced write activity

	hard disk	flash SSD
Transactions/sec	864	3045
CPU utilization (%)	20	65
Log write size (sectors)	32	30
Log write time (msec)	8.1	1.3

Again the overall performance is I/O limited with the hard disk and CPU limited with flash SSD. Demonstrated by increasing compute power using quad cores instead of dual cores. Same performance for hard disk and 35% gain with flash SSD

# Impact on use of temporary tables

- Temporary tables are often swapped in external sorting and during joins (sort merge and hash)
- External sort: produce small runs of the size of available buffer, sequential write them to persistent storage, many random reads in merge phase → ideal for flash SSD
- Disk requires large clusters (64 KB) with fewer random accesses during merge, flash SSD performs better with small clusters (2KB) and many random accesses
- External sorting of 2 000 000 tuples (200 MB)
  - Required approx. 155 sec w. hard disk
  - Required approx. 30 sec w. flash SSD



# Hash anomaly?

- Hashing is supposed to be the dual of sorting in terms of access patterns: many short writes (while partitioning) and few long reads (while processing buckets)
- This suggests that hashing will not perform much better on flash SSD
- Results proved differently, could be because of implementation
- However, sort merge join might have a revival under flash SSD for temporary tables.



# The 5 minute rule revisited

- Gray and Putzolu stated the 5 minute rule in 1987
- 5 minute rule calculates the break even point where it is cheaper to keep data in memory vs. reading it in again
  - Often used data should be kept in memory
  - Seldom used data should be read from disk
- Based on costs of main memory and disks in 1987, the break even point was around 400 seconds (rounded to 5 minutes)
- $\text{BreakEvenIntSecs} = (\text{PagesPerMBofRAM}/\text{AccPerSecPerDisk}) \times (\text{PricePerDiskDrive} / \text{PricePerMBofRAM})$



# 5 Minute Rule revisited

- Under today's prices for RAM, disks and using the original 1KB page the break even point would be 6 hours
- Using 4 KB pages the break even is around 90 minutes (1.5 hrs)
- Using 32 GB flash SSD at \$999 the break even point would be around 15 minutes
- Using 2009 prices and capacities we are already back to the original 5 minutes
- With today's prices we dropped to about 2 minutes

