

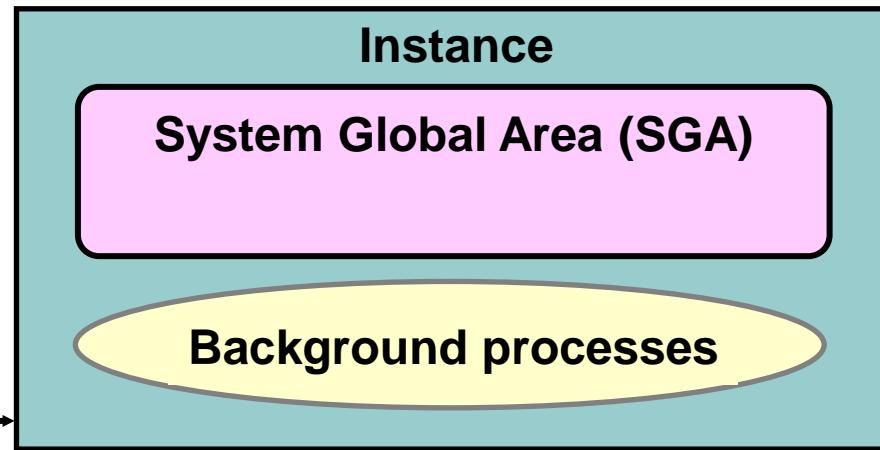
Oracle Database Architecture

- An Oracle server:
 - Is a database management system that provides an open, comprehensive, integrated approach to information management
 - Consists of an **Oracle instance** and an **Oracle database**



Database Structures

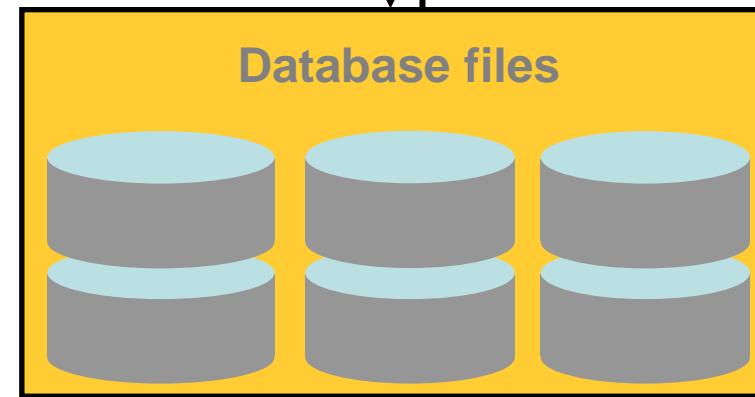
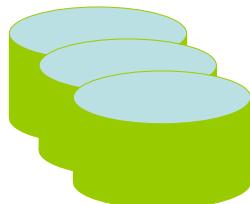
Memory structures



Process structures



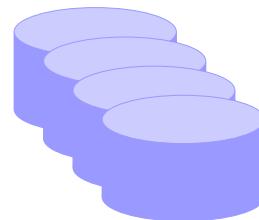
Storage structures



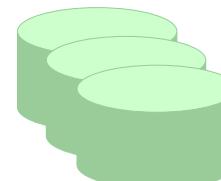
Physical Database Structure



Control files



• Data files



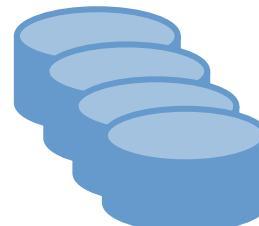
• Online redo log files



• Parameter file



• Password file



• Backup files



• Archive log files



• Alert and trace log files

Files of a database

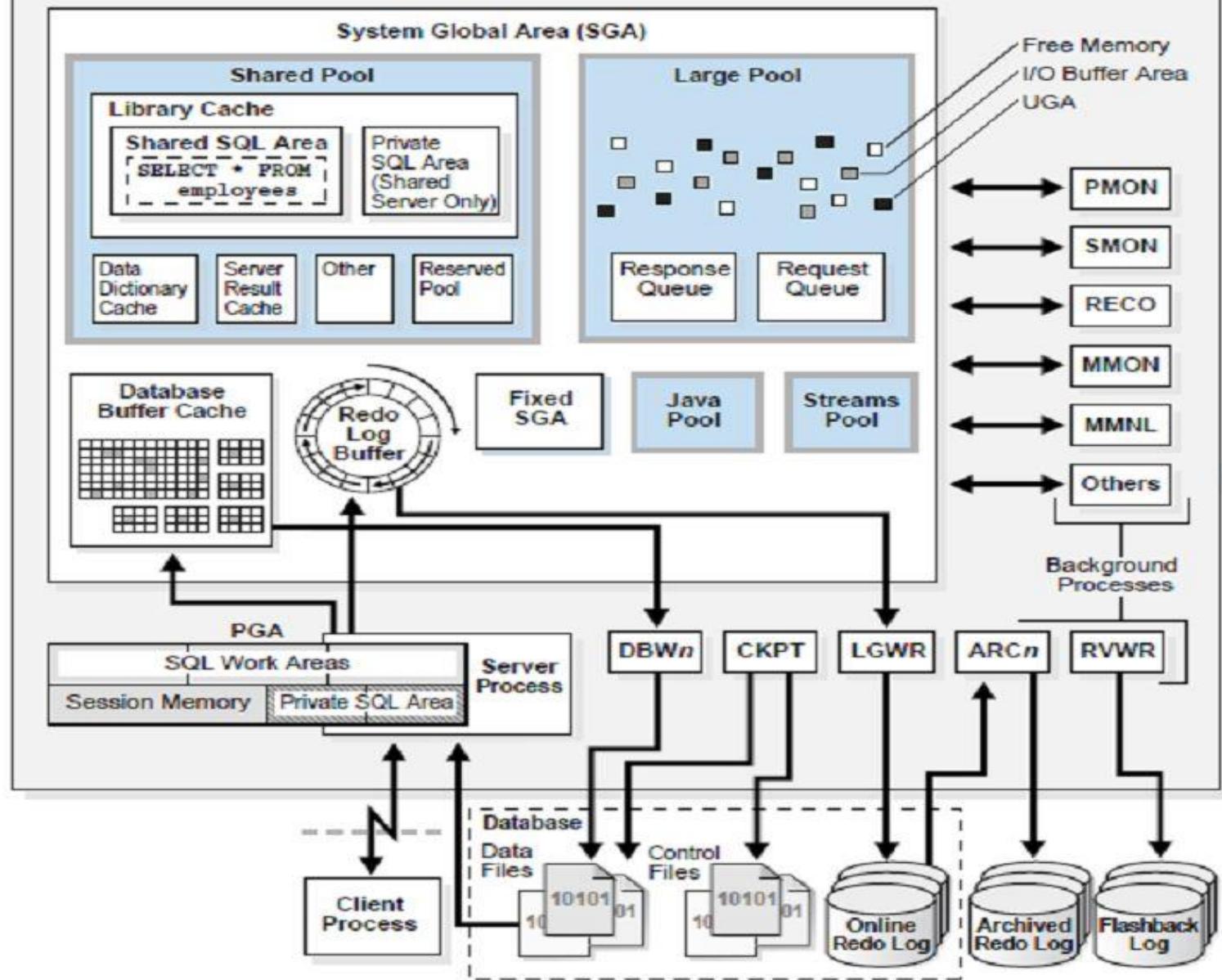
A **data file** is a physical file on disk that was created by Oracle Database and contains data structures such as tables and indexes.

A **control file** contains information such as the following: the database name, information about data files, online redo log files, tablespace information, etc.

The **online redo log** is a set of files containing records of changes made to data. Online redo log is the most crucial structure for recovery.

Alert log is a file that provides a chronological log of database messages and errors.

Instance



Data Dictionary Views

	Who Can Query	Contents	Subset of	Notes
DBA_	DBA	Everything	N/A	May have additional columns meant for DBA use only
ALL_	Everyone	Everything that the user has privileges to see	DBA_views	Includes user's own objects
USER_	Everyone	Everything that the user owns	ALL_views	Is usually the same as ALL_ except for the missing OWNER column. Some views have abbreviated names as PUBLIC synonyms.

Data Dictionary: Usage Examples

a

```
SELECT table_name, tablespace_name FROM  
user_tables;
```

b

```
SELECT sequence_name, min_value, max_value,  
increment_by FROM all_sequences WHERE  
sequence_owner IN ('MDSYS', 'XDB');
```

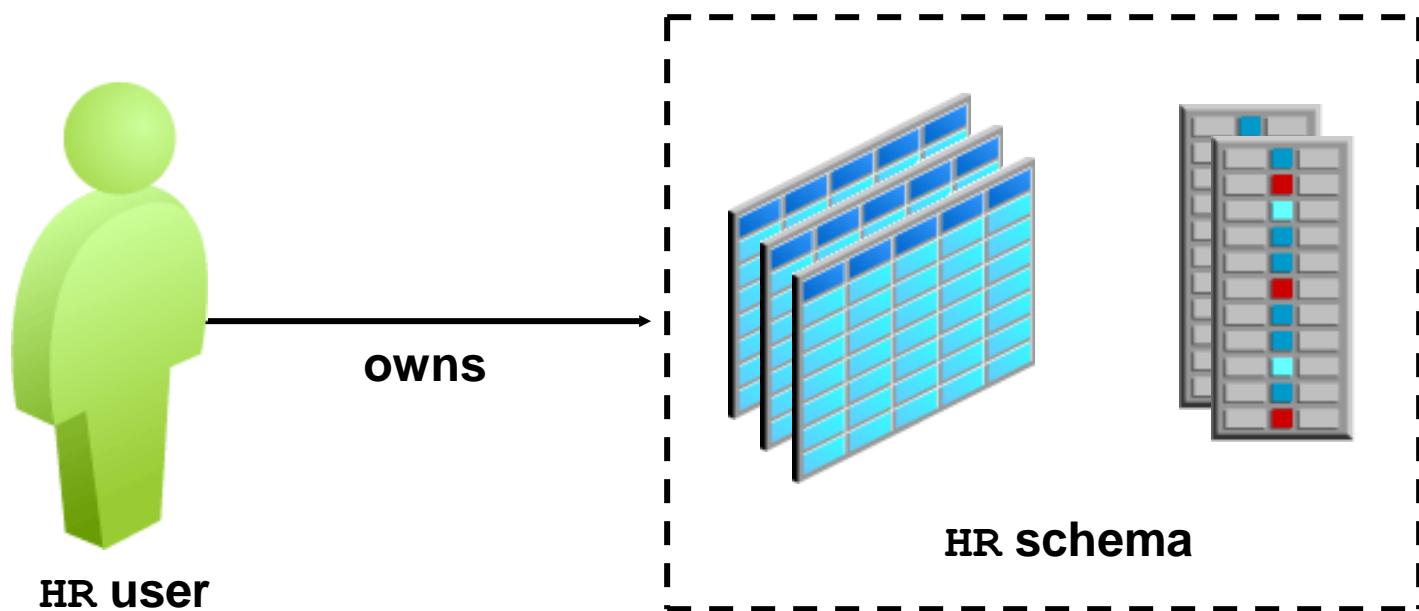
c

```
SELECT USERNAME, ACCOUNT_STATUS FROM  
dba_users WHERE ACCOUNT_STATUS = 'OPEN';
```

d

```
DESCRIBE dba_indexes;
```

What Is a Schema?



Schema Objects

- In Oracle Database, a database **schema** is a collection of logical data structures, or **schema objects**. A database schema is owned by a database user and has the same name as the **username**.

Accessing Schema Objects

Database Instance: orcl.oracle.com

Home Performance **Administration** Maintenance



Schema

Database Objects	Programs	XML Database
Tables	Packages	Configuration
Indexes	Package Bodies	Resources
Views	Procedures	Access Control Lists
Synonyms	Functions	XML Schemas
Sequences	Triggers	XMLType Tables
Database Links	Java Classes	XMLType Views
Directory Objects	Java Sources	
Reorganize Objects		

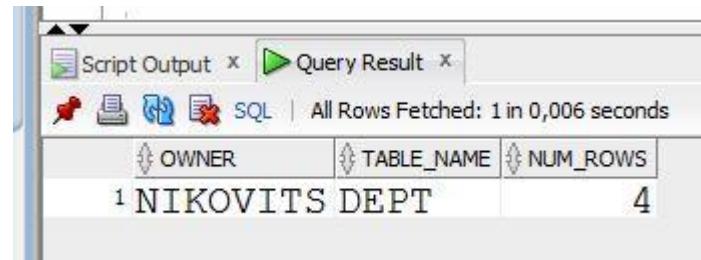
Users & Privileges	Materialized Views	BI & OLAP
Users	Materialized Views	Dimensions
Roles	Materialized View Logs	Cubes
Profiles	Refresh Groups	OLAP Dimensions
Audit Settings		Measure Folders

Tables

```
CREATE TABLE dept
(deptno NUMBER(2), dname VARCHAR2(42), loc VARCHAR2(39));
```

```
SELECT owner, table_name, num_rows
FROM DBA_TABLES
WHERE owner='NIKOVITS' AND table_name='DEPT';
```

(!) ANALYZE TABLE DEPT COMPUTE STATISTICS;
(!) ANALYZE TABLE DEPT DELETE STATISTICS;



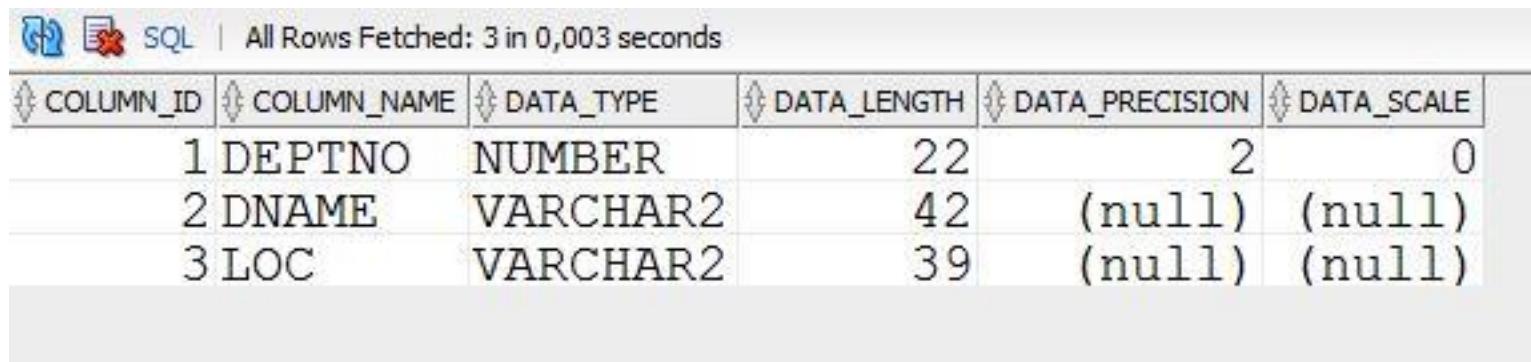
The screenshot shows the Oracle SQL Developer interface with the 'Query Result' tab selected. The results of the query are displayed in a table:

OWNER	TABLE_NAME	NUM_ROWS
1 NIKOVITS	DEPT	4

Tables

```
CREATE TABLE dept
  (deptno NUMBER(2), dname VARCHAR2(42), loc VARCHAR2(39));

SELECT column_id, column_name, data_type, data_length,
       data_precision, data_scale
  FROM DBA_TAB_COLUMNS
 WHERE owner='NIKOVITS' AND table_name='DEPT';
```

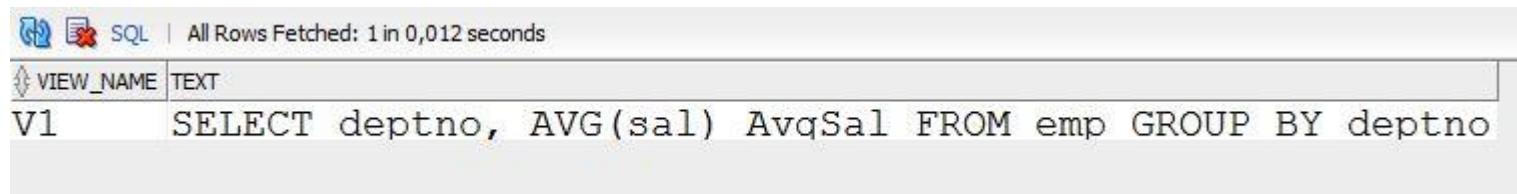


COLUMN_ID	COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION	DATA_SCALE
1	DEPTNO	NUMBER	22	2	0
2	DNAME	VARCHAR2	42	(null)	(null)
3	LOC	VARCHAR2	39	(null)	(null)

Views

```
CREATE VIEW v1 AS
SELECT deptno, AVG(sal) AvgSal FROM emp GROUP BY deptno;
```

```
SELECT view_name, text
FROM DBA_VIEWS
WHERE owner='NIKOVITS' AND view_name='V1';
```



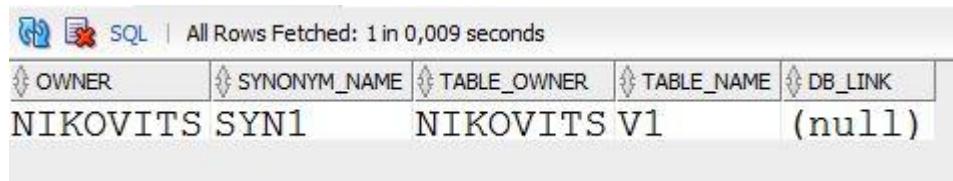
The screenshot shows a SQL query results window. The title bar says 'SQL | All Rows Fetched: 1 in 0,012 seconds'. The table has two columns: 'VIEW_NAME' and 'TEXT'. The 'VIEW_NAME' column contains 'V1' and the 'TEXT' column contains the SQL code for the view.

VIEW_NAME	TEXT
V1	SELECT deptno, AVG(sal) AvgSal FROM emp GROUP BY deptno

Synonyms

```
CREATE SYNONYM syn1 FOR v1;
```

```
SELECT * FROM DBA_SYNONYMS  
WHERE owner='NIKOVITS' AND synonym_name='SYN1';
```



OWNER	SYNONYM_NAME	TABLE_OWNER	TABLE_NAME	DB_LINK
NIKOVITS	SYN1	NIKOVITS	V1	(null)

```
SELECT * FROM syn1 WHERE deptno > 10;
```

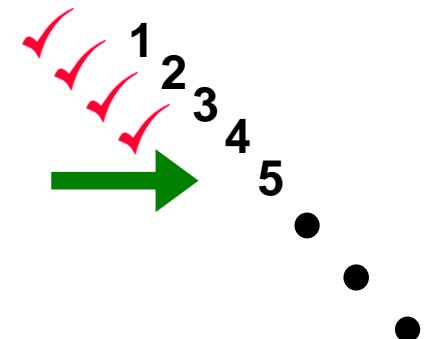


DEPTNO	AVGSAL
30	1800
20	2175

Sequences

- A sequence is a mechanism for automatically generating integers that follow a pattern.

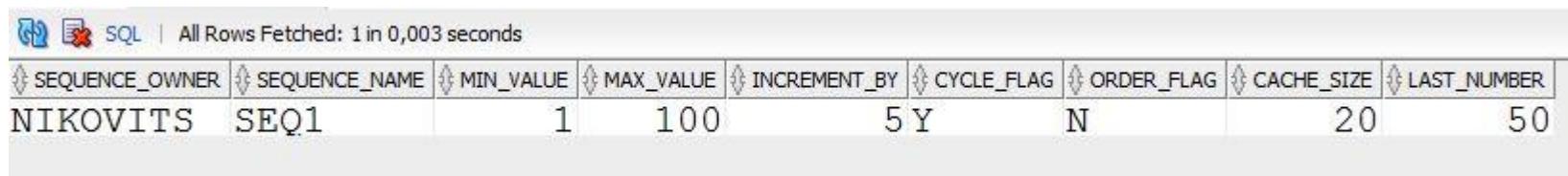
- A sequence has a name, which is how it is referenced when the next value is requested.
- A sequence is not associated with any particular table or column.
- The progression can be ascending or descending.
- The interval between numbers can be of any size.
- A sequence can cycle when a limit is reached.



Sequences

```
CREATE SEQUENCE seq1
MINVALUE 1 MAXVALUE 100 INCREMENT BY 5
START WITH 50 CYCLE;
```

```
SELECT * FROM DBA_SEQUENCES
WHERE sequence_name='SEQ1';
```



The screenshot shows a SQL developer interface with the following details:

- Toolbar icons: Refresh, Undo, Redo, SQL (highlighted in blue), and a red asterisk.
- Text: "All Rows Fetched: 1 in 0,003 seconds"
- Table structure:

SEQUENCE_OWNER	SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	CYCLE_FLAG	ORDER_FLAG	CACHE_SIZE	LAST_NUMBER
NIKOVITS	SEQ1	1	100	5	Y	N	20	50

Using a Sequence

Next value from sequence:

```
INSERT INTO dept VALUES(seq1.NEXTVAL, 'IT', 'Budapest');
```

Current value from sequence:

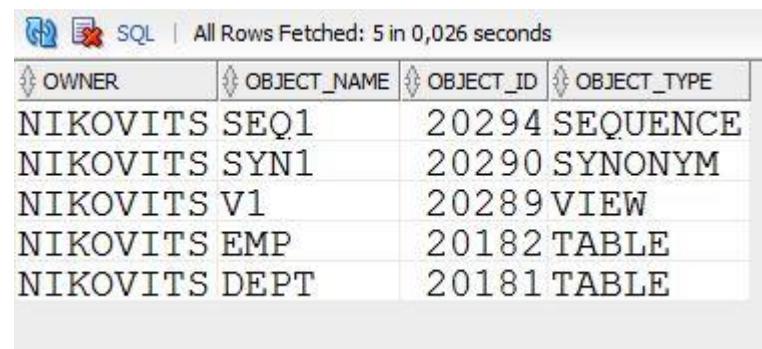
```
INSERT INTO emp(deptno, empno, ename, job, sal)  
VALUES(seq1.CURRVAL, 1, 'Tailor', 'SALESMAN', 100);
```

Current value from sequence:

```
INSERT INTO emp(deptno, empno, ename, job, sal)  
VALUES(seq1.CURRVAL, 2, 'Sailor', 'SALESMAN', 200);
```

ANY Object

```
SELECT owner, object_name, object_id, object_type  
FROM DBA_OBJECTS  
WHERE owner='NIKOVITS', and created > sysdate - 1;
```



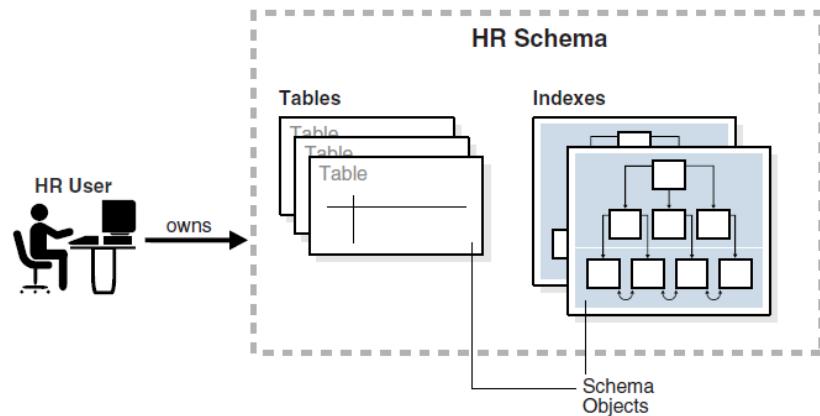
The screenshot shows a SQL query results window. The title bar says "SQL | All Rows Fetched: 5 in 0,026 seconds". The results table has four columns: OWNER, OBJECT_NAME, OBJECT_ID, and OBJECT_TYPE. The data is as follows:

OWNER	OBJECT_NAME	OBJECT_ID	OBJECT_TYPE
NIKOVITS	SEQ1	20294	SEQUENCE
NIKOVITS	SYN1	20290	SYNONYM
NIKOVITS	V1	20289	VIEW
NIKOVITS	EMP	20182	TABLE
NIKOVITS	DEPT	20181	TABLE

Oracle database concepts

A database **schema** is a logical container for data structures, called **schema objects**. Examples of schema objects are tables and indexes

Figure 2-1 HR Schema



The most important schema objects in a relational database are tables. A **table** stores data in rows.

Oracle SQL enables you to create and manipulate many other types of schema objects, including the following: Indexes, Views, Sequences, Synonyms, PL/SQL subprograms etc.

Some schema objects store data in logical storage structures called **segments**. For example, a nonpartitioned heap-organized **table or an index** creates a segment. Other schema objects, such as **views** and **sequences**, consist of **metadata only**.

At the operating system level, Oracle Database stores database data in **data files**. For ease of administration, Oracle Database allocates space for user data in **tablespaces**, which like **segments** are **logical storage structures**. Each segment belongs to only one tablespace. For example, the data for a nonpartitioned table is stored in a single segment, which is turn is stored in one tablespace.

A permanent tablespace contains persistent schema objects. Objects in permanent tablespaces are stored in data files. A **temporary tablespace** contains schema objects only for the duration of a session.

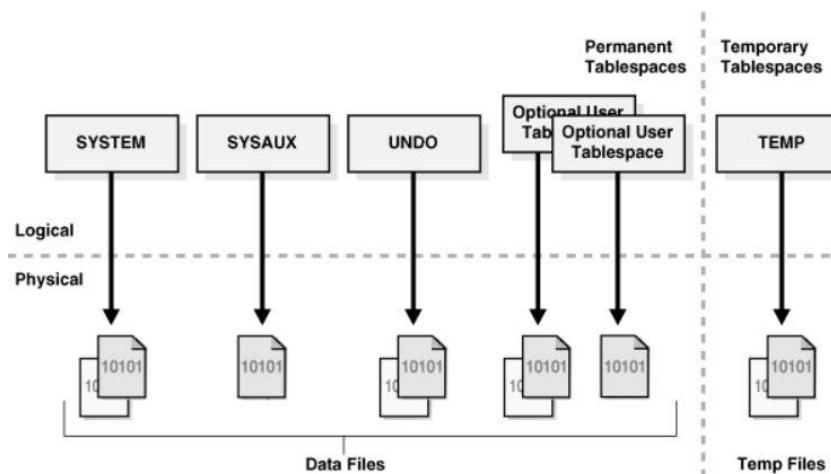
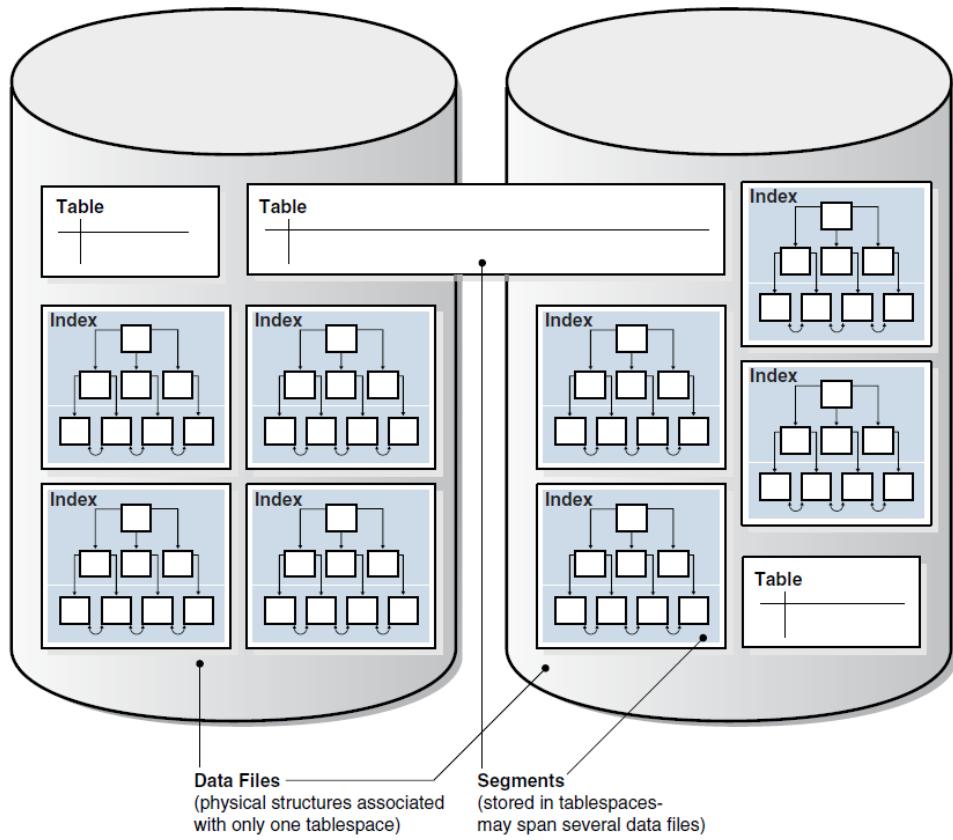


Figure 2–2 Segments, Tablespaces, and Data Files



View

A view is a logical representation of one or more tables. In essence, **a view is a stored query**. A view derives its data from the tables on which it is based, called base tables. Base tables can be tables or other views. All operations performed on a view actually affect the base tables. **You can use views in most places where tables are used.**

Unlike a table, a view is not allocated storage space, nor does a view contain data. Rather, a view is defined by a query that extracts or derives data from the base tables referenced by the view. Because a view is based on other objects, it **requires no storage** other than storage for the query that defines the view in the data dictionary.

Sequence

A sequence is a schema object from which multiple **users can generate unique integers**. A sequence generator provides a highly scalable and well-performing method to generate surrogate keys for a number data type.

The sequence generator is useful in multiuser environments for generating unique numbers without the overhead of disk I/O or transaction locking. For example, two users simultaneously insert new rows into the orders table. By using a sequence to generate unique numbers for the order_id column, neither user has to wait for the other to enter the next available order number. The sequence automatically generates the correct values for each user.

Synonym

A synonym is an alias for a schema object. For example, you can create a synonym **for a table or view, sequence, PL/SQL program unit, user-defined object type, or another synonym**. Because a synonym is simply an alias, it requires no storage other than its definition in the data dictionary. Synonyms can simplify SQL statements for database users. Synonyms are also useful for hiding the identity and location of an underlying schema object. If the underlying **object** must be **renamed or moved**, then only the synonym must be redefined. Applications based on the synonym continue to work without modification.

Data Dictionary

The data dictionary base tables are the first objects created in any Oracle database. All data dictionary tables and views for a database are **stored in the SYSTEM tablespace**. Because the SYSTEM tablespace is always online when the database is open, the data dictionary is always available when the database is open.

The Oracle Database user **SYS owns** all base tables and user-accessible **views of the data dictionary**. Data in the base tables of the data dictionary is necessary for Oracle Database to function.

During database operation, Oracle Database reads the data dictionary to ascertain that schema objects exist and that users have proper access to them. Oracle Database also updates the data dictionary continuously to **reflect changes in database structures**, auditing, grants, and data.

For example, **if user hr creates a table** named interns, then new rows are added to the data dictionary that reflect the new table, columns, segment, extents, and the privileges that hr has on the table. This new information is visible the next time the dictionary views are queried.

Oracle Database creates **public synonyms** for many data dictionary views so users can access them conveniently

Physical and Logical Storage

An Oracle **database is a set of files** that store Oracle data in persistent disk storage.

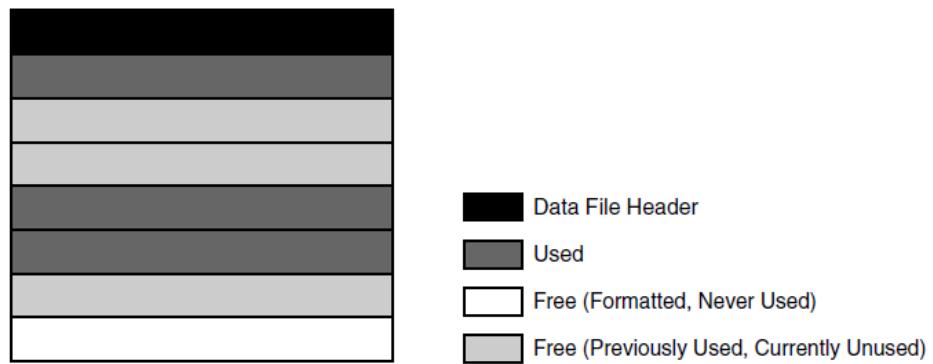
A **data file** is a physical file on disk that was created by Oracle Database and contains data structures such as tables and indexes. A **temp file** is a data file that belongs to a temporary tablespace. The data is written to these files in an Oracle proprietary format that cannot be read by other programs.

Oracle Database creates a data file for a **tablespace** by allocating the specified amount of disk space plus the overhead for the data file header.

When Oracle Database first creates a data file, the allocated disk space is formatted but **contains no user data**. However, the database reserves the space to hold the data for future **segments** of the associated tablespace. As the data grows in a tablespace, Oracle Database uses the free space in the data files to allocate **extents** for the segment.

The following figure illustrates the different types of space in a data file. **Extents are either used**, which means they contain segment data, **or free**, which means they are available for reuse. Over time, updates and deletions of objects within a tablespace can create pockets of empty space that individually are not large enough to be reused for new data. This type of empty space is called *fragmented free space*.

Figure 11–5 Space in a Data File



Oracle Database allocates logical space for all data in the database. The logical units of database space allocation are data blocks, extents, segments, and tablespaces. At a physical level, the data is stored in data files on disk. The data in the data files is stored in **operating system blocks**.

An operating system block is the **minimum unit of data that the operating system can read or write**. In contrast, an **Oracle block** is a logical storage structure whose size and structure are not known to the operating system.

Figure 12-6 Data Blocks and Operating System Blocks

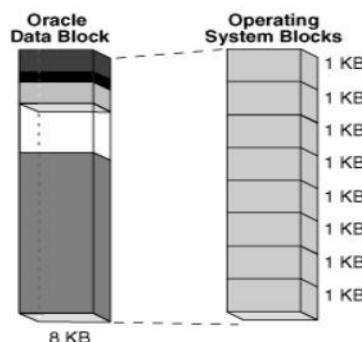
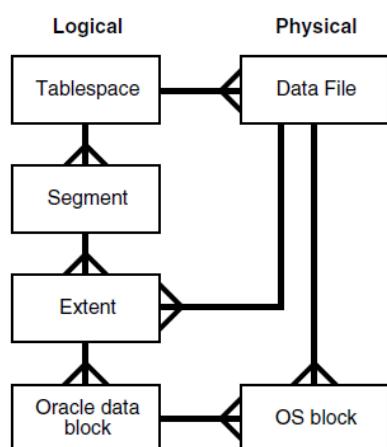


Figure 12–1 is an entity-relationship diagram for physical and logical storage. The crow's foot notation represents one-to-many relationship.

Figure 12–1 Logical and Physical Storage

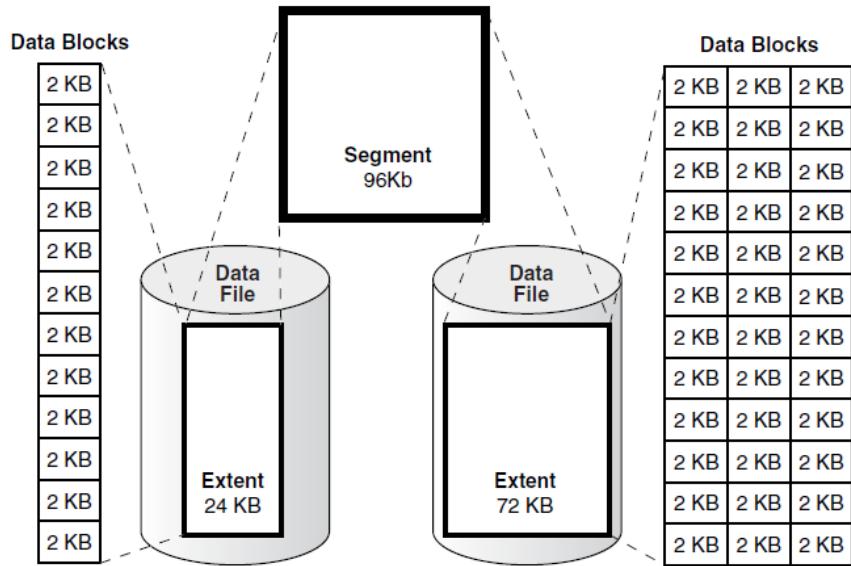


At the finest level of granularity, Oracle Database stores data in **data blocks**. One logical data block corresponds to a specific number of bytes of physical disk space, for example, 2 KB. Data blocks are the **smallest units of storage that Oracle Database can use** or allocate.

An **extent** is a set of logically **contiguous data blocks** allocated for storing a specific type of information.

A **segment** is a **set of extents** allocated for a specific database object, such as a table. For example, the data for the employees table is stored in its own data segment, whereas each index for employees is stored in its own index segment. Every database object that consumes storage consists of a single segment.

Figure 12–2 Segments, Extents, and Data Blocks Within a Tablespace



Logical Space Management

Oracle Database must use logical space management to track and allocate the extents in a tablespace. When a database object requires an extent, the database must have a method of finding and providing it. Similarly, when an object no longer requires an extent, the database must have a method of making the free extent available.

Oracle Database manages space within a tablespace based on the type that you create. You can create either of the following types of tablespaces:

- Locally managed tablespaces (default)
- Dictionary-managed tablespaces

A **dictionary-managed tablespace** uses the data dictionary to manage its extents.

A **locally managed tablespace** maintains a bitmap in the data file header to track free and used space in the data file body. Each bit corresponds to a group of blocks. When space is allocated or freed, Oracle Database changes the bitmap values to reflect the new status of the blocks.

Within a locally managed tablespace, the database can manage segments automatically or manually.

Automatic Segment Space Management (ASSM) avoids the need to manually determine correct settings for many storage parameters. Only one crucial SQL parameter controls space allocation: **PCTFREE**. This parameter specifies the percentage of space to be reserved in a block for future updates.

The legacy **Manual Segment Space Management** (MSSM) method uses a linked list called a free list to manage free space in the segment. In addition to PCTFREE, MSSM requires you to control space allocation with SQL parameters such as PCTUSED, FREELISTS, and FREELIST GROUPS.

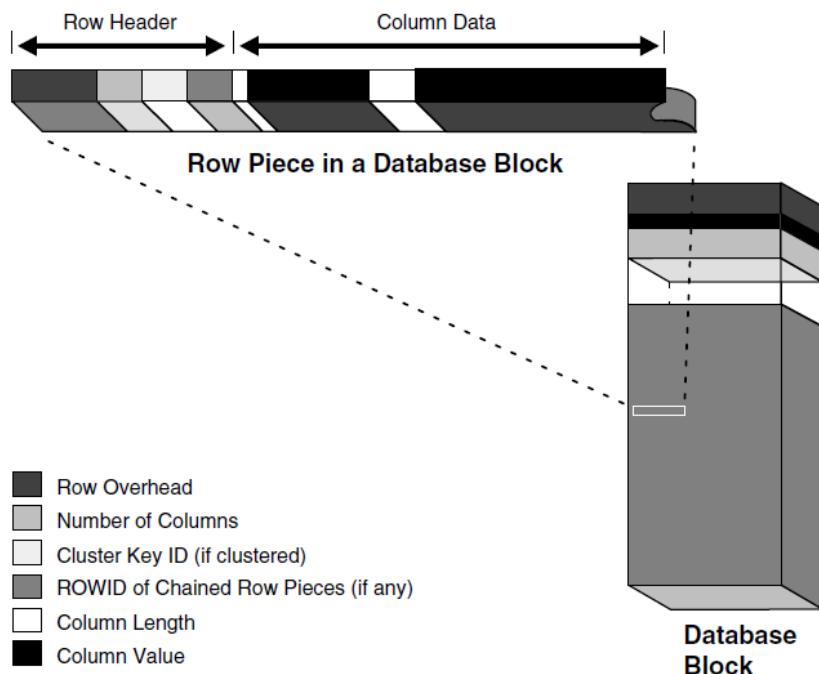
Block size

Every database has a database block size. The **DB_BLOCK_SIZE** initialization parameter sets the data block size for a database when it is created. The size is set for the **SYSTEM** and **SYSAUX tablespaces** and is the default for all other tablespaces. The database block size cannot be changed except by re-creating the database.

You can create **individual tablespaces whose block size differs** from the **DB_BLOCK_SIZE** setting.

Oracle Database stores rows as **variable-length records**. A row is contained in one or more row pieces. Each row piece has a row header and column data.

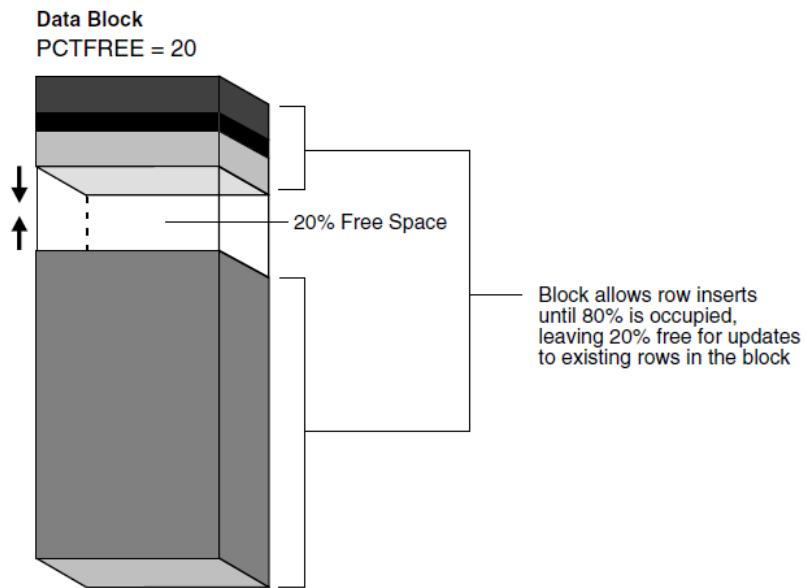
Figure 12-7 The Format of a Row Piece



As the database fills a data block from the bottom up, the amount of free space between the row data and the block header decreases. This free space can also shrink during updates, as when changing a trailing null to a nonnull value. The database manages free space in the data block to optimize performance and avoid wasted space.

The **PCTFREE** storage parameter is essential to how the database manages free space. This SQL parameter sets the minimum percentage of a data block reserved as free space for updates to existing rows.

Figure 12–9 PCTFREE



A single data segment in a database stores the data for one user object. There are **different types of segments**. Examples of user segments include:

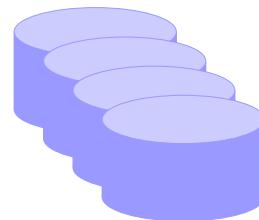
Table, table partition, table cluster, LOB or LOB partition, Index or index partition.

By default, the database uses **deferred segment creation** to update only database metadata when creating tables and indexes. When a user inserts the first row into a table or partition, the database creates segments for the table or partition, its LOB columns, and its indexes.

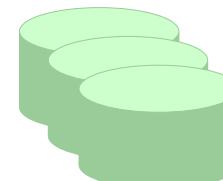
Physical Database Structure



Control files



• Data files



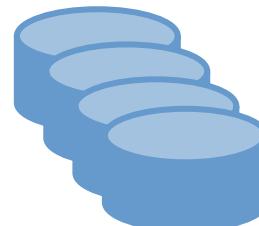
• Online redo log files



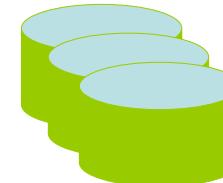
• Parameter file



• Password file



• Backup files



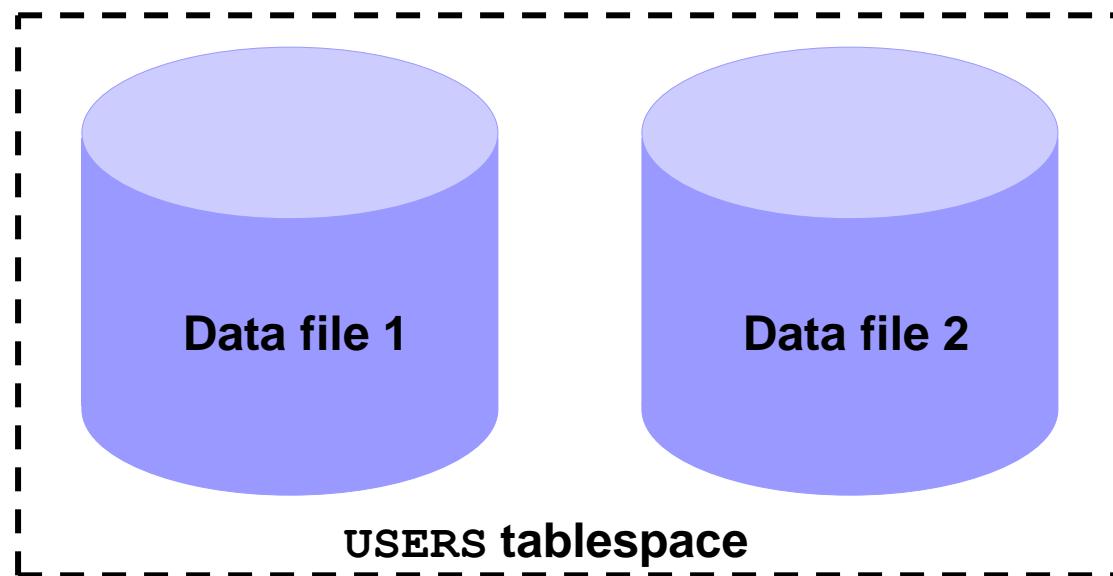
• Archive log files



• Alert and trace log files

Tablespaces and Data Files

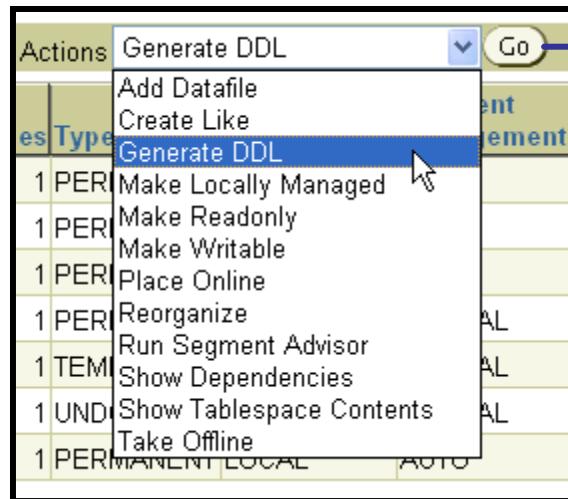
- Tablespaces consist of one or more data files.
- Data files belong to only one tablespace.



SYSTEM and SYSAUX Tablespaces

- The SYSTEM and SYSAUX tablespaces are mandatory tablespaces.
- They are created at the time of database creation.
- They must be online.
- The SYSTEM tablespace is used for core functionality (for example, data dictionary tables).
- The auxiliary SYSAUX tablespace is used for additional database components (such as the Enterprise Manager Repository).

Actions with Tablespaces



Show DDL

[Return](#)

```
CREATE SMALLFILE TABLESPACE "EXAMPLE" DATAFILE
  '/u01/app/oracle/oradata/orcl/example01.dbf' SIZE 100M REUSE AUTOEXTEND ON
  NEXT 640K MAXSIZE 32767M NOLOGGING EXTENT MANAGEMENT LOCAL SEGMENT SPACE
  MANAGEMENT AUTO
```

Dropping Tablespaces



A screenshot of a database management interface showing a list of tablespaces. The "EXAMPLE" tablespace is selected, indicated by a blue radio button in the "Select" column. A blue arrow points from the "Delete" button in the warning dialog to the "Delete" button in the table's toolbar. The table has the following columns: Select, Name, Size (MB), Used (MB), Used (%), Free (MB), Status, Datafiles, Type, Extent Management, and Segment Management. The data for the selected "EXAMPLE" tablespace is: Size 100.0 MB, Used 68.2 MB, Used 68.2%, Free 68.2 MB, Status OK, Datafiles 1, Type PERMANENT, Extent Management LOCAL, Segment Management AUTO.

Select	Name	Size (MB)	Used (MB)	Used (%)	Free (MB)	Status	Datafiles	Type	Extent Management	Segment Management
<input checked="" type="radio"/>	EXAMPLE	100.0	68.2	68.2	31.8	✓	1	PERMANENT	LOCAL	AUTO
<input type="radio"/>	INVENTORY	5.0	0.1	2	1.2	4.9	✓	1	PERMANENT	LOCAL
<input type="radio"/>	SYSAUX	240.0	237.2	98.8	2.8	✓	1	PERMANENT	LOCAL	AUTO
<input type="radio"/>	SYSTEM	470.0	468.1	99.6	1.9	✓	1	PERMANENT	LOCAL	MANUAL
<input type="radio"/>	TEMP	20.0	0.0	0	20.0	✓	1	TEMPORARY	LOCAL	MANUAL
<input type="radio"/>	UNDOTBS1	35.0	9.6	27.3	25.4	✓	1	UNDO	LOCAL	MANUAL
<input type="radio"/>	USERS	5.0	3.0	60.0	2.0	✓	1	PERMANENT	LOCAL	AUTO

Viewing Tablespace Information

```
SELECT tablespace_name, status, contents, logging, extent_management,  
allocation_type, segment_space_management  
FROM dba_tablespaces
```



TABLESPACE_NAME	STATUS	CONTENTS	LOGGING	EXTENT_MAN	ALLOCATIO	SEGMENT
SYSTEM	ONLINE	PERMANENT	LOGGING	LOCAL	SYSTEM	MANUAL
UNDOTBS1	ONLINE	UNDO	LOGGING	LOCAL	SYSTEM	MANUAL
SYSAUX	ONLINE	PERMANENT	LOGGING	LOCAL	SYSTEM	AUTO
TEMP	ONLINE	TEMPORARY	NOLOGGING	LOCAL	UNIFORM	MANUAL
USERS	ONLINE	PERMANENT	LOGGING	LOCAL	SYSTEM	AUTO
EXAMPLE	ONLINE	PERMANENT	NOLOGGING	LOCAL	SYSTEM	AUTO
INVENTORY	ONLINE	PERMANENT	LOGGING	LOCAL	SYSTEM	AUTO

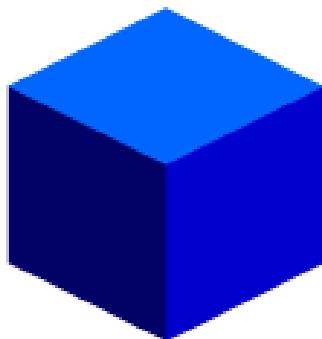
```
SELECT ts#, name FROM v$tablespace
```



TS#	NAME
0	SYSTEM
1	UNDOTBS1
2	SYSAUX
4	USERS
3	TEMP
6	EXAMPLE
7	INVENTORY

Segments, Extents, and Blocks

- Segments exist within a tablespace.
- Segments are made up of a collection of extents.
- Extents are a collection of data blocks.
- Data blocks are mapped to disk blocks.



Segment



Extents

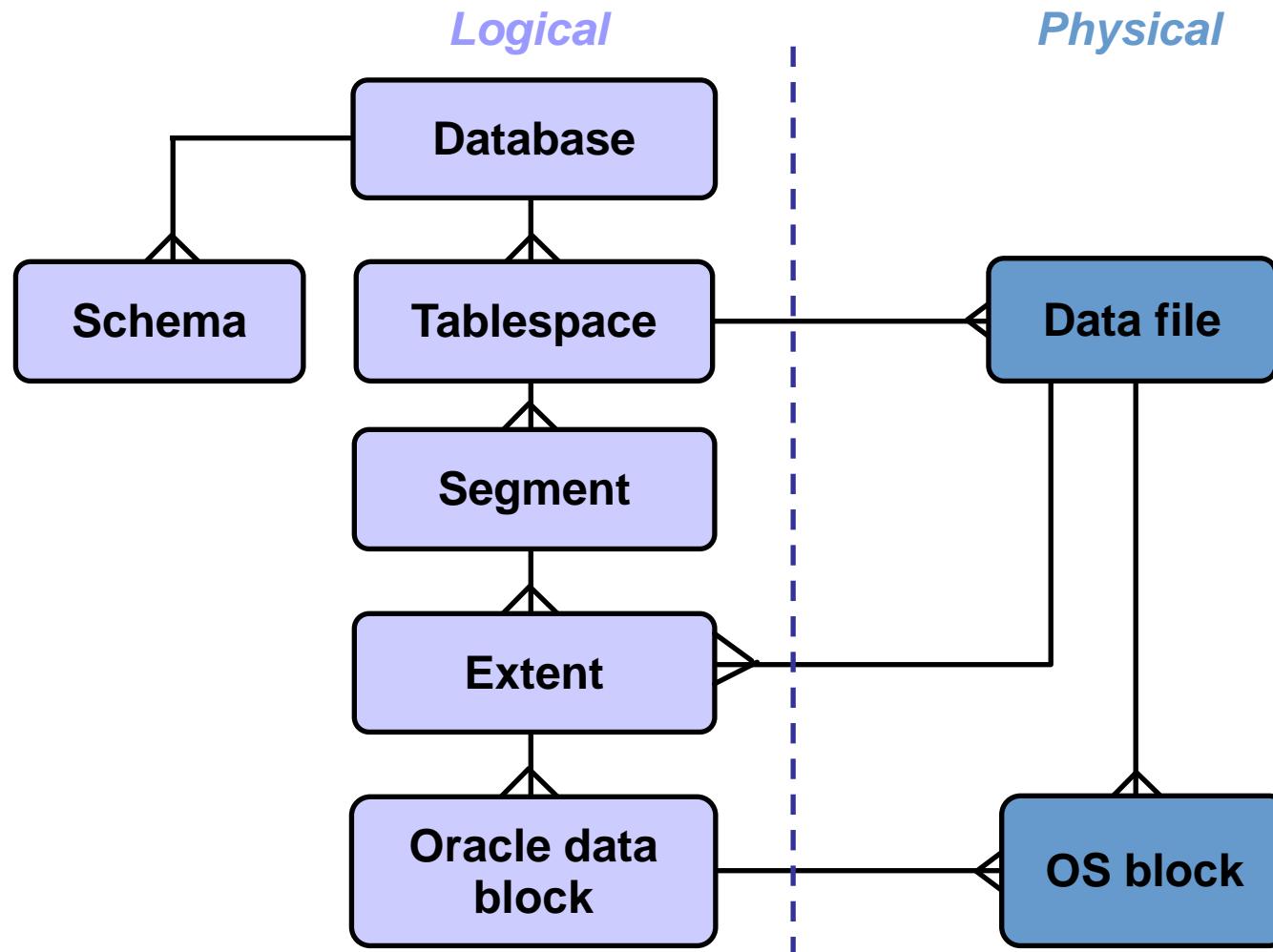


Data blocks



Disk blocks

Logical and Physical Database Structures



Viewing Tablespace Contents

Database Instance: EDRSR10P1_orcl.us.oracle.com > Tablespaces > View Tablespace: EXAMPLE > Show Tablespace Contents

Show Tablespace Contents

Size (MB)	100.0	Used (MB)	68.3	Extent Mgmt	LOCAL	Auto Extend	Yes
Block Size (KB)	8	Used (%)	68.3	Segment Mgmt	AUTO	Extents	836

Segments

Search

Segment Name	Type	Minimum Size (KB)	Minimum Extents
<input type="text"/>	<input type="button" value="All Types"/>	<input type="text"/>	<input type="button" value="Go"/>

You can use the wildcard symbol (%) in the segment name.

Previous 1-10 of 418 Next 10

Segment Name	Type	Size (KB)	Extents
SH.CUSTOMERS	TABLE	12,288	27
SH.SUPPLEMENTARY_DEMOGRAPHICS	TABLE	4,096	19
OE.PRODUCT_DESCRIPTIONS	TABLE	3,072	18
SH.SALES.SALES_Q4_2001	TABLE PARTITION	2,048	17
SH.SALES.SALES_Q3_2001		1,024	16
SH.SALES.SALES_Q1_1999		1,024	16
SH.CUSTOMERS_PK		1,024	16
SH.SALES.SALES_Q2_2001		960	15
SH.SALES.SALES_Q1_2001		960	15
SH.SALES.SALES_Q1_2000		960	15

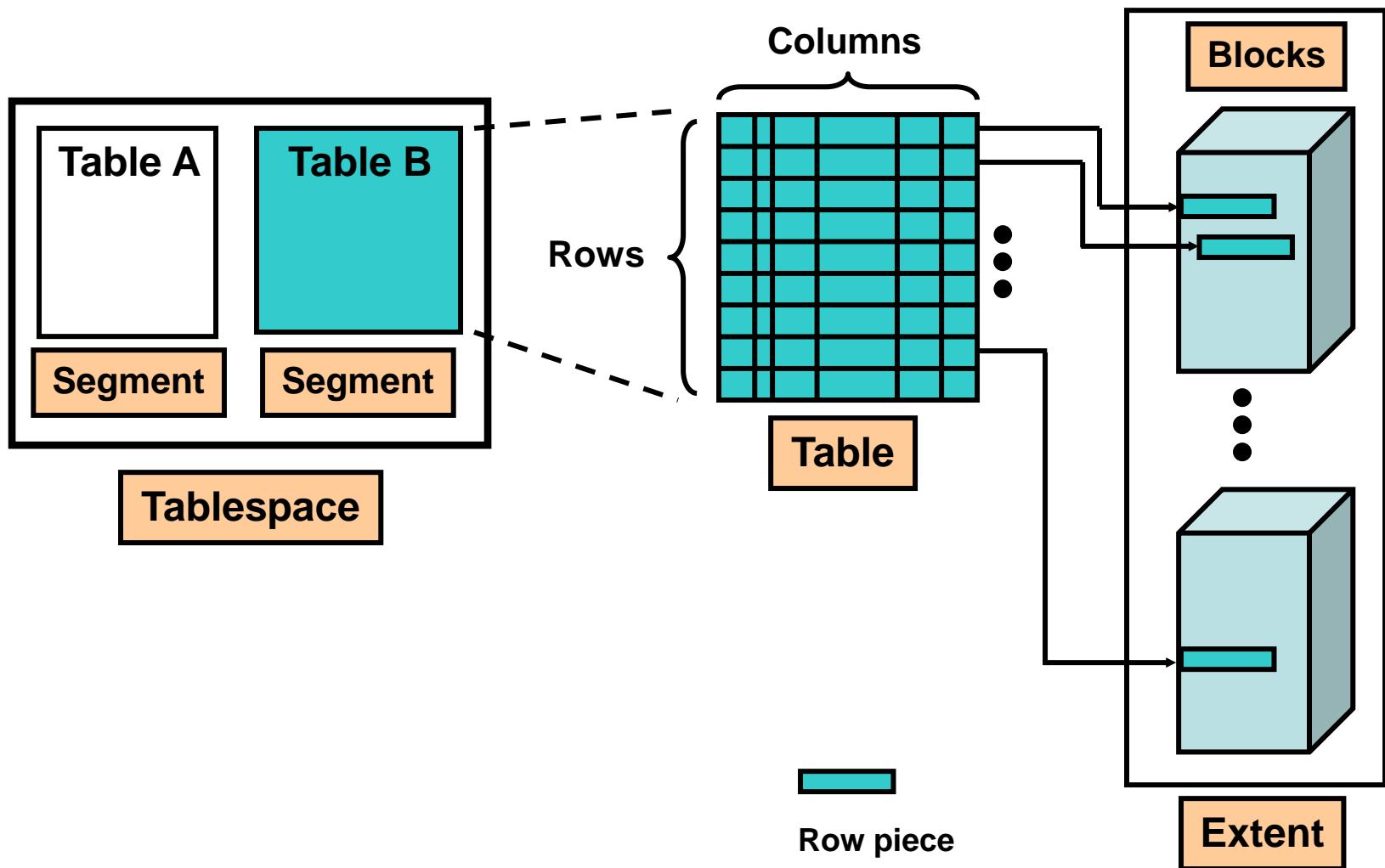
Extent Map

Clicking the Highlight Extents button on the Extent Map. Clicking on a used extent in the Extent Map.

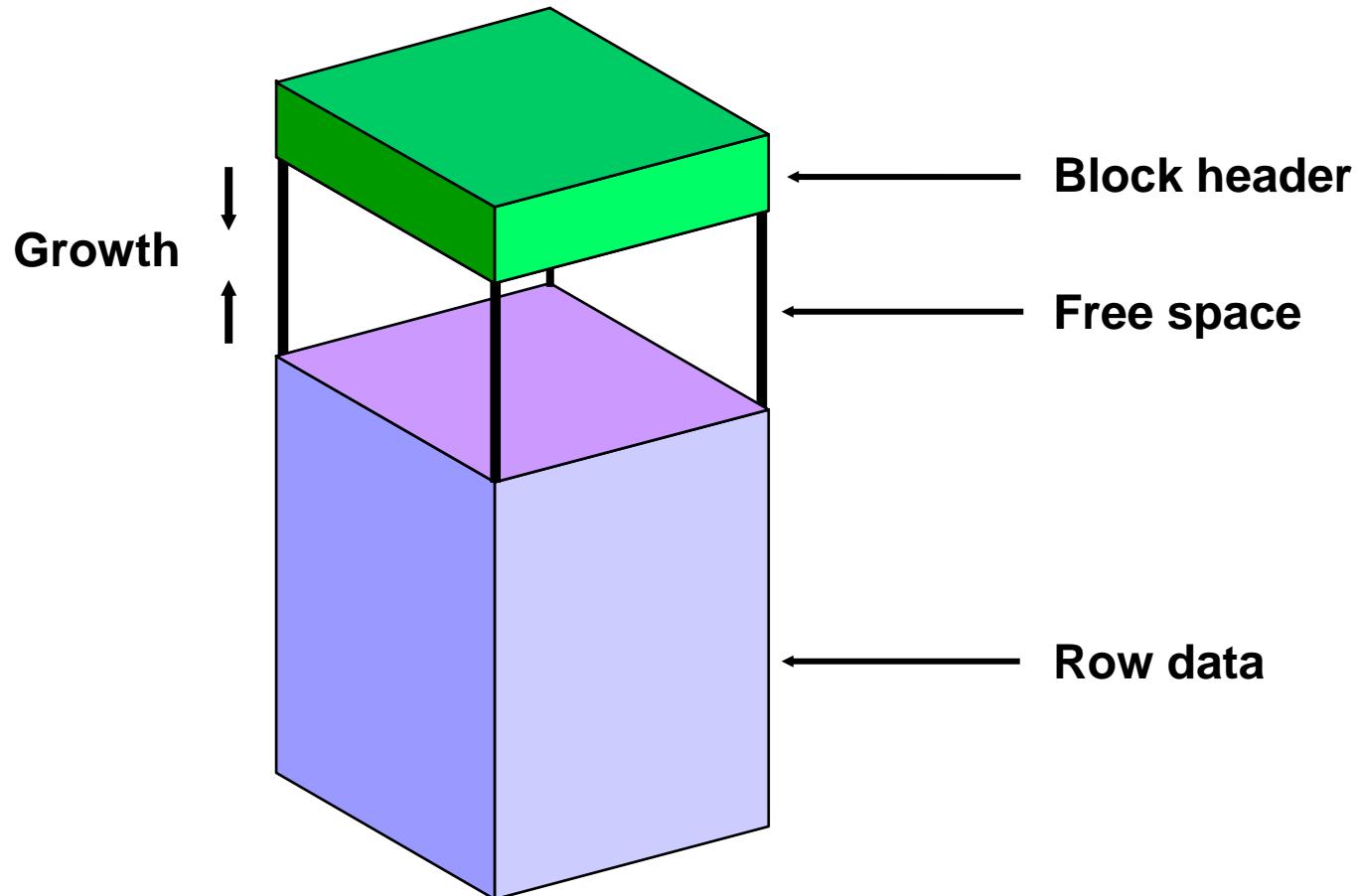
- Header
- Used
- Free
- Selected
- Unmapped

Next 10

How Table Data Is Stored

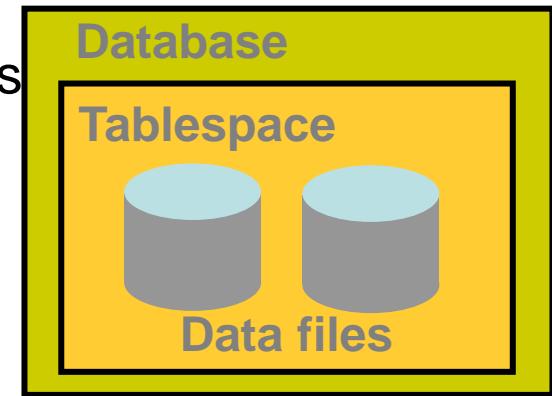


Anatomy of a Database Block



Tablespaces and Data Files

- The Oracle database stores data logically in tablespaces and physically in data files.
 - Tablespaces:
 - Can belong to only one database
 - Consist of one or more data files
 - Are further divided into logical units
 - Data files:
 - Can belong to only one tablespace and one database
 - Are a repository for schema object data



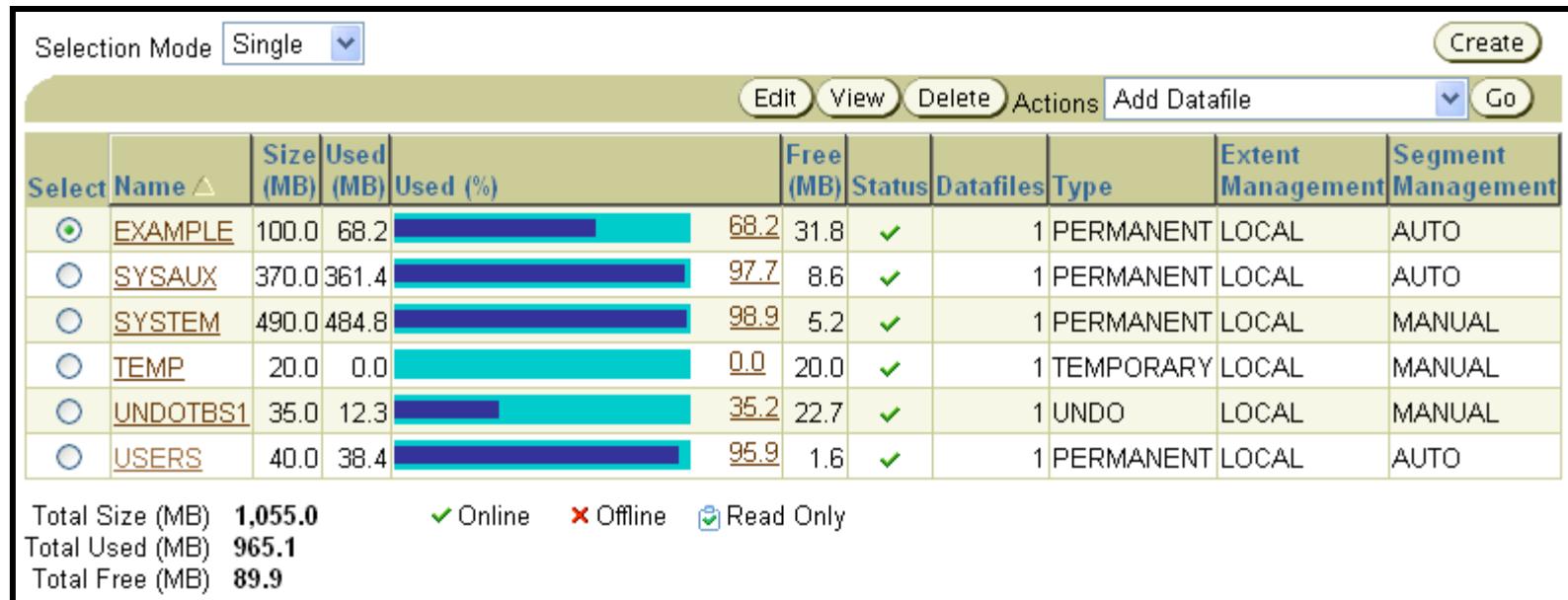
Space Management in Tablespaces

- Locally managed tablespace:
 - Free extents are managed in the tablespace.
 - A bitmap is used to record free extents.
 - Each bit corresponds to a block or group of blocks.
 - The bit value indicates free or used extents.
 - The use of locally managed tablespaces is recommended.
- Dictionary-managed tablespace:
 - Free extents are managed by the data dictionary.
 - Appropriate tables are updated when extents are allocated or unallocated.
 - These tablespaces are supported only for backward compatibility.



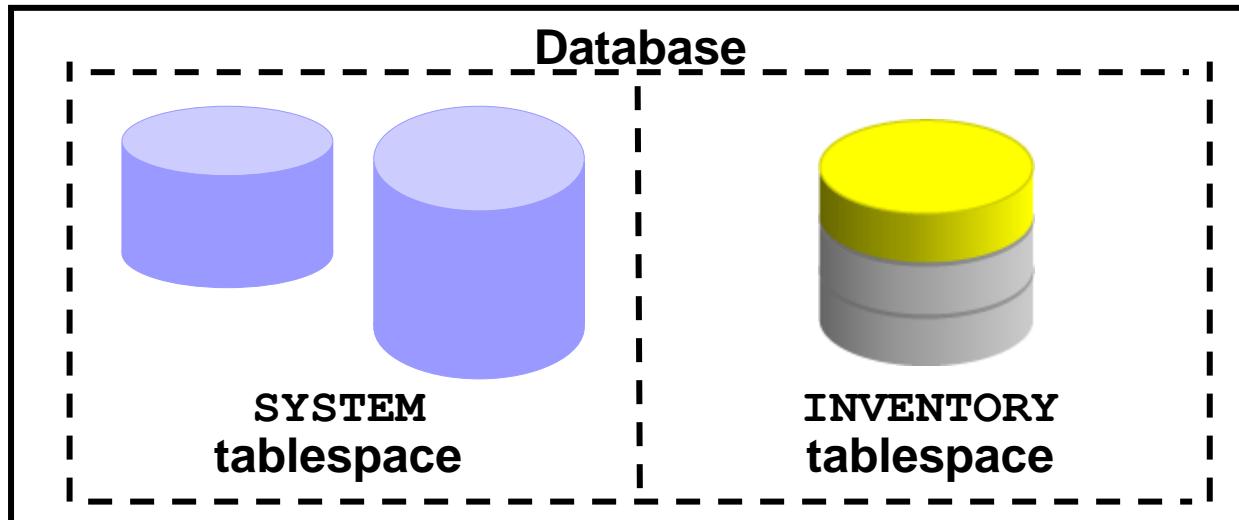
Tablespaces in the Preconfigured Database

- SYSTEM
- SYSAUX
- TEMP
- UNDOTBS1
- USERS
- EXAMPLE



Enlarging the Database

- You can enlarge the database in the following ways:
 - Creating a new tablespace
 - Adding a data file to an existing tablespace
 - Increasing the size of a data file
 - Providing for the dynamic growth of a data file



Oracle database concepts

One characteristic of an RDBMS is the **independence** of logical data structures such as tables, views, and indexes **from physical** storage structures.

Because physical and logical structures are separate, you can manage physical storage of data without affecting access to logical structures. For example, renaming a database file does not rename the tables stored in it.

A **data file** is a physical file on disk that was created by Oracle Database and contains data structures such as tables and indexes. A **temp file** is a data file that belongs to a temporary tablespace. The database writes data to these files in an Oracle proprietary format that cannot be read by other programs.

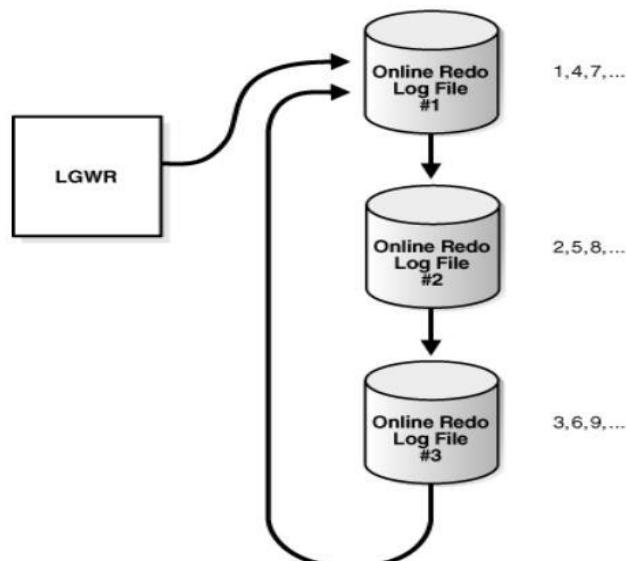
The database **control file** is a small binary file associated with only one database. Each database has one unique control file, although multiple identical copies are permitted. A control file contains information such as the following: the database name, information about data files, online redo log files, tablespace information, etc. The control file contains information required to recover the database, including checkpoints. A **checkpoint** indicates the SCN (System Change Number) in the redo stream where instance recovery would be required to begin.

The most crucial structure for recovery is the **online redo log**. The online redo log is a set of files **containing records of changes** made to data. The database maintains online redo log files to protect against data loss. Specifically, after an instance failure, the online redo log files enable Oracle Database to recover committed data that it has not yet written to the data files.

Server processes write every transaction synchronously to the **redo log buffer**, which the LGWR process then writes to the online redo log. Contents of the online redo log include uncommitted transactions, and schema and object management statements.

Log writer writes to online redo log files **circularly**. A **log switch** occurs when the database stops writing to one online redo log file and begins writing to another. Normally, a switch occurs when the current online redo log file is full and writing must continue.

Figure 11-6 Reuse of Online Redo Log Files



Most Oracle databases store files in a **file system**, which is a data structure built inside a contiguous disk address space. All operating systems have file managers that allocate and deallocate disk space into files within a file system. A file system enables disk space to be allocated to many files. Each file has a name and is made to appear as a contiguous address space to applications such as Oracle Database. The database can create, read, write, resize, and delete files.

Oracle Managed Files is a file naming strategy that enables you to specify operations in terms of database objects rather than file names. For example, you can create a tablespace without specifying the names of its data files.

With **user-managed files**, you directly manage the operating system files in the database. You make the decisions regarding file structure and naming. For example, when you create a tablespace you set the name and path of the tablespace data files.

Oracle Managed Files does not eliminate existing functionality. You can create new files while manually administering old files. Thus, a database can have a mixture of Oracle Managed Files and user-managed files.

At the operating system level, Oracle Database stores database data in structures called **data files**. Every Oracle database must have at least one data file.

A database must have the **SYSTEM** and **SYSAUX** tablespaces. Oracle Database automatically allocates the first data files of any database for the SYSTEM tablespace during database creation.

The SYSTEM tablespace contains the **data dictionary**, a set of tables that contains database metadata. Typically, a database also has an **undo tablespace** and a temporary tablespace (usually named TEMP).

A **permanent tablespace** contains persistent schema objects. Objects in permanent tablespaces are stored in data files.

A **temporary tablespace** contains schema objects only for the duration of a session. Locally managed temporary tablespaces have temporary files (**temp files**), which are special files designed to **store data in hash, sort, and other operations**. Temp files also store result set data when insufficient space exists in memory.

When Oracle Database first creates a data file, the allocated disk space is formatted but contains no user data. However, the database reserves the space to hold the data for future segments of the associated tablespace. As the data grows in a tablespace, Oracle Database uses the free space in the data files to allocate extents for the segment.

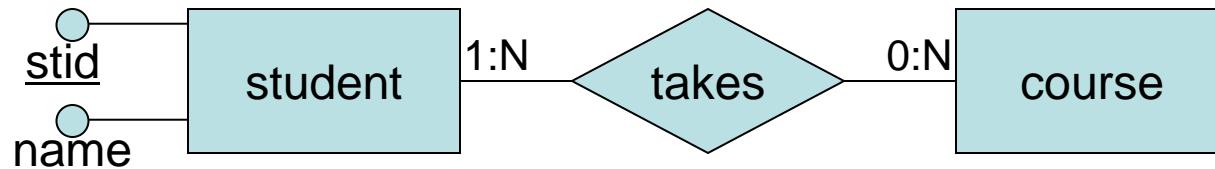
Physical Storage Organization

Outline

- Where and How data are stored?
 - physical level
 - logical level

Building a Database: **High-Level**

- Design conceptual schema using a data model, e.g. ER, UML, etc.



Building a Database: Logical-Level

- Design logical schema, e.g. relational, network, hierarchical, object-relational, XML, etc schemas
- Data Definition Language (DDL)

```
CREATE TABLE student
(cid char(8) primary key, name varchar(32))
```

student	
<u>cid</u>	name

Populating a Database

- Data Manipulation Language (**DML**)

INSERT INTO student VALUES ('00112233', 'Paul')

student	
<u>cid</u>	name
00112233	Paul

Transaction operations

- **Transaction**: a collection of operations performing a single logical function

```
BEGIN TRANSACTION transfer
```

```
    UPDATE bank-account SET balance = balance - 100 WHERE account=1
```

```
    UPDATE bank-account SET balance = balance + 100 WHERE account=2
```

```
COMMIT TRANSACTION transfer
```

- A failure during a transaction can leave system in an inconsistent state, e.g. transfers between bank accounts.

Where and How all this information is stored?

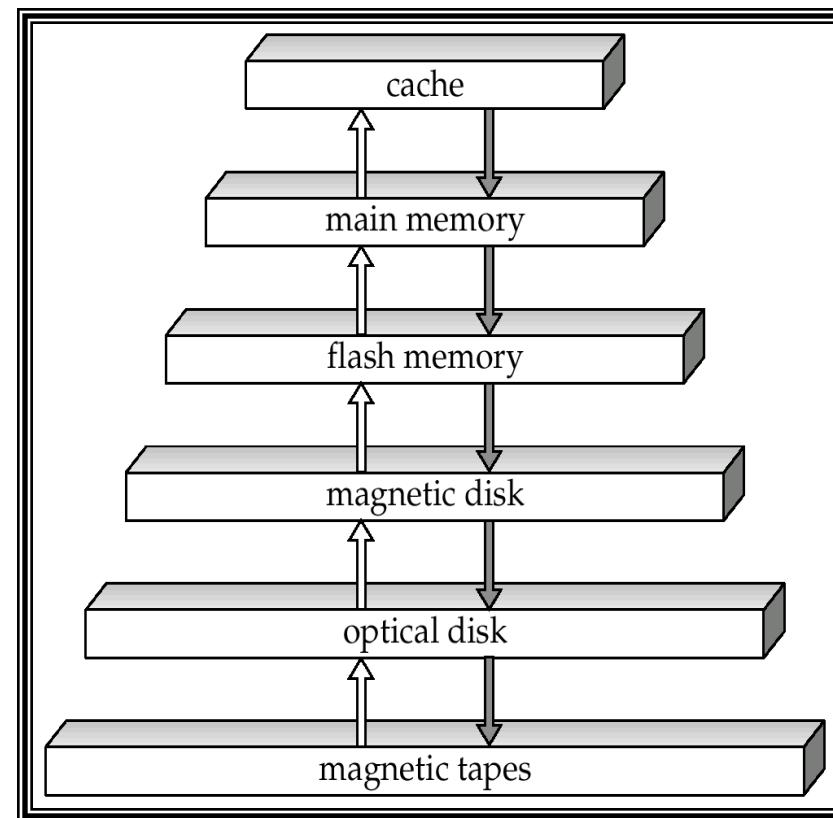
- **Metadata**: tables, attributes, data types, constraints, etc
- Data: records
- Transaction logs, indices, etc

Where: In Main Memory?

- Fast!
- But:
 - Too small
 - Too expensive
 - Volatile

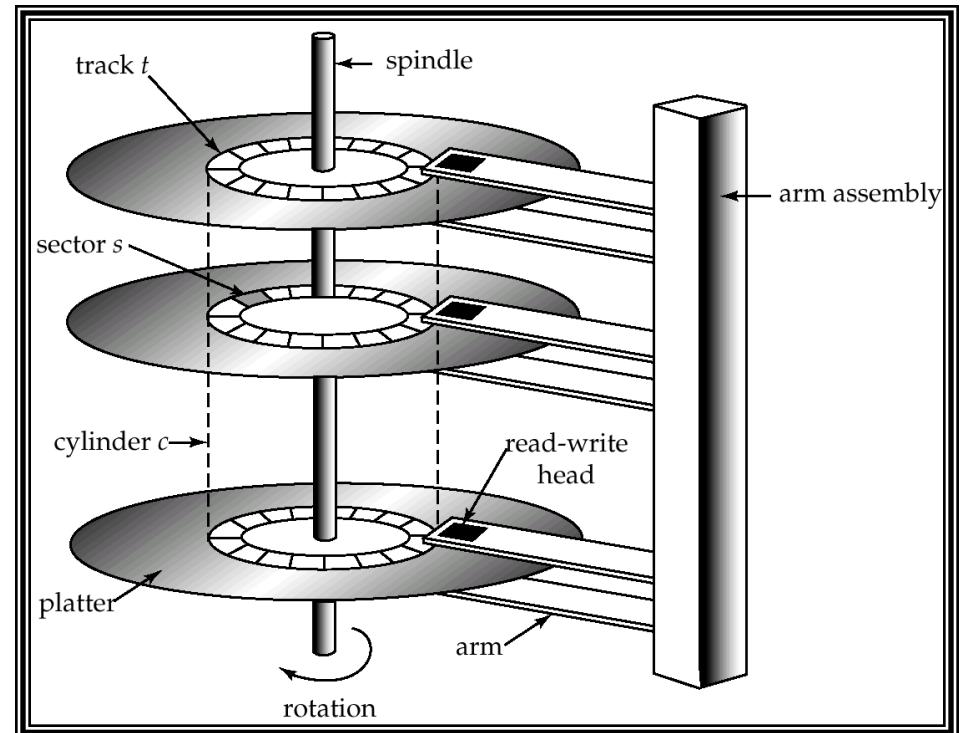
Physical Storage Media

- Primary Storage
 - Cache
 - Main memory
- Secondary Storage
 - Flash memory
 - Magnetic disk
- Offline Storage
 - Optical disk
 - Magnetic tape



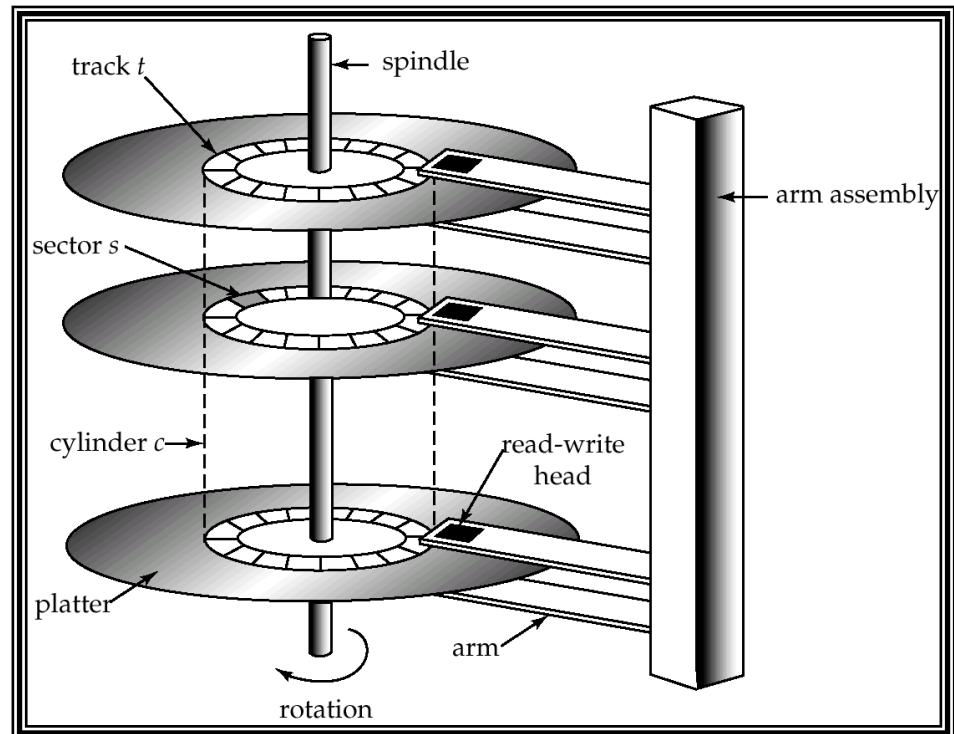
Magnetic Disks

- Random Access
- Inexpensive
- Non-volatile



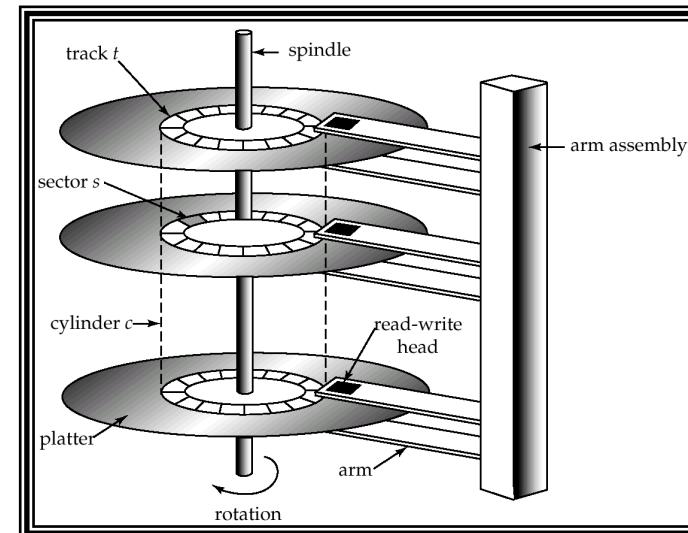
How do disks work?

- Platter: covered with magnetic recording material
- Track: logical division of platter surface
- Sector: hardware division of tracks
- Block: OS division of tracks
 - Typical block sizes:
512 B, 2KB, 4KB
- Read/write head



Disk I/O

- Disk I/O := block I/O
 - Hardware address is converted to Cylinder, Surface and Sector number
 - Modern disks: Logical Sector Address 0...n
- **Access time**: time from read/write request to when **data transfer begins**
 - **Seek time**: the head reaches correct track
 - Average seek time 5-10 msec
 - **Rotation latency time**: correct block rotated under head
 - 5400 RPM, 15K RPM
 - On average 4-11 msec
- **Block Transfer Time**



Optimize I/O

- Database system performance **I/O bound**
- **Improve the speed of access** to disk:
 - Scheduling algorithms (elevator algorithm)
 - File Organization (heap, index, hash)
- Introduce disk redundancy
 - Redundant Array of Independent Disks (**RAID**)
- **Reduce number of I/Os**
 - Query optimization, indexes

~~Where~~ and How all this information is stored?

- Metadata: tables, attributes, data types, constraints, etc
- Data: records
- Transaction logs, indices, etc
- A collection of **files (or tables)**
 - **Physically** partitioned into **pages or data blocks**
 - **Logically** partitioned into **records**

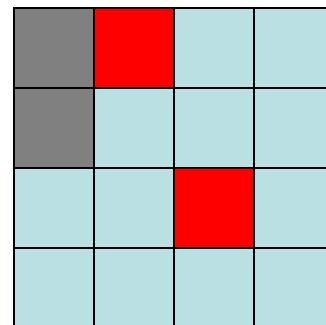
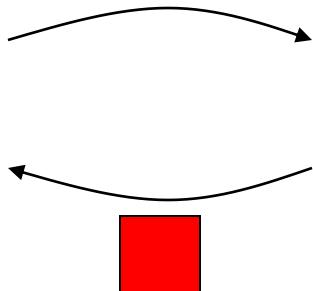
Storage Access

- A collection of files
 - Physically partitioned into pages
 - Typical database page sizes: 2KB, 4KB, 8KB
 - Reduce number of block I/Os := reduce number of page I/Os
 - How?
- **Buffer Manager**

Buffer Management (1/2)

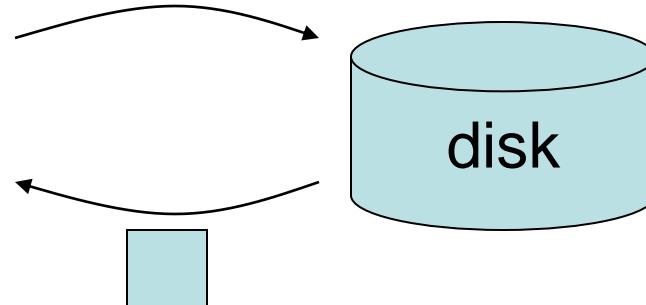
- Buffer: storing a page copy
- Buffer manager: manages a pool of buffers
 - Requested page in pool: hit!
 - Requested page in disk:
 - Allocate page frame
 - Read page and pin
- Problems?

Page request



buffer pool

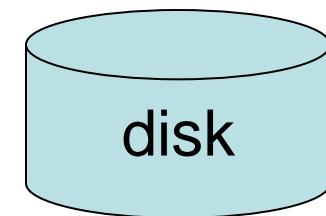
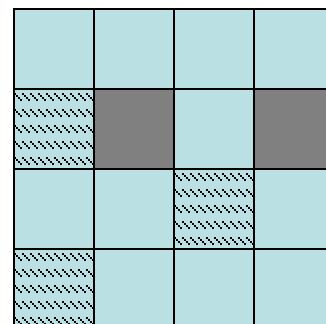
Page request



Buffer Management (2/2)

- What if **no empty page frame** exists:
 - Select victim page
 - Each page associated with **dirty flag**
 - If page selected dirty, then write it back to disk
- Which page to select?
 - Replacement policies (**LRU**, MRU)

Page request



Disk Arrays

- Single disk becomes bottleneck
- Disk arrays
 - instead of single large disk
 - many small **parallel disks**
 - read N blocks in a single access time
 - concurrent queries
 - tables spanning among disks
- Redundant Arrays of Independent Disks (**RAID**)
 - 7 levels (0-6)
 - reliability
 - **redundancy**
 - **parallelism**

RAID Technology

- A natural solution is a large array of small independent disks acting as a single higher-performance logical disk.
- A concept called **data striping** is used, which utilizes parallelism to improve disk performance.
- Data striping distributes data transparently over multiple disks to make them appear as a single large, fast disk.

Figure 17.13

Striping of data across multiple disks.

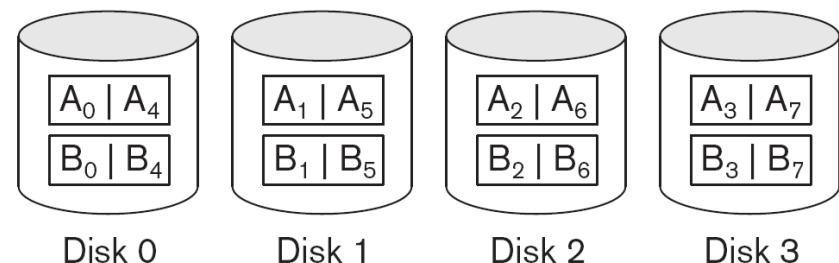
(a) Bit-level striping across four disks.

(b) Block-level striping across four disks.

A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7
B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7

(a)

Data

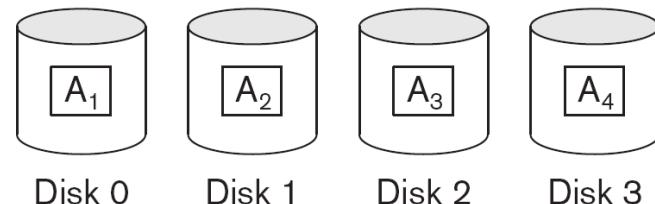


Disk 0 Disk 1 Disk 2 Disk 3

File A:

Block A ₁	Block A ₂	Block A ₃	Block A ₄
----------------------	----------------------	----------------------	----------------------

(b)



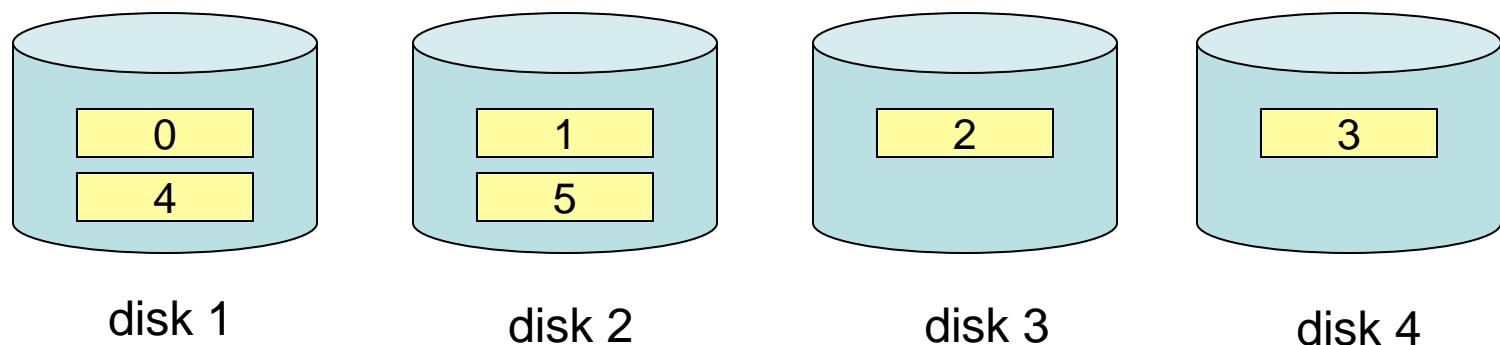
Disk 0 Disk 1 Disk 2 Disk 3

RAID Technology (cont.)

- Different raid organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information.
 - Raid **level 0** (**striping**) has no redundant data and hence has the best write performance at the risk of data loss
 - Raid **level 1** uses mirrored disks.
 - Raid **level 2** uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Level 2 includes both error detection and correction.
 - Raid **level 3** uses a single parity disk relying on the disk controller to figure out which disk has failed.
 - Raid **Levels 4 and 5** use block-level data striping, with level 5 distributing data and parity information across all disks.
 - Raid **level 6** applies the so-called P + Q (two parity) redundancy scheme using Reed-Solomon codes to protect against up to two disk failures by using just two redundant disks.

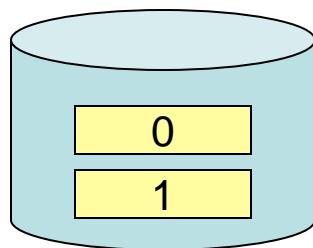
RAID level 0

- Block level **striping**
- No redundancy
- maximum bandwidth
- automatic load balancing
- **best write performance**
- **but, no reliability**

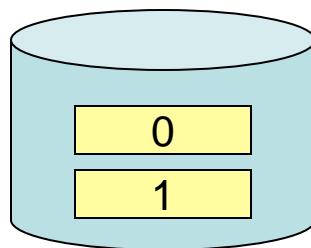


Raid level 1

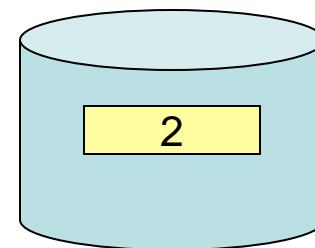
- **Mirroring**
 - Two identical copies stored in two different disks
- **Parallel reads**
- **Sequential writes**
- transfer rate comparable to single disk rate
- **most expensive** solution



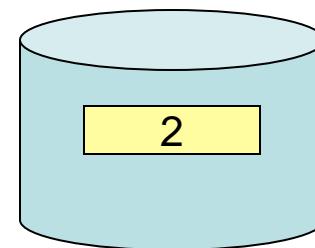
disk 1



disk 2
mirror of disk 1



disk 3



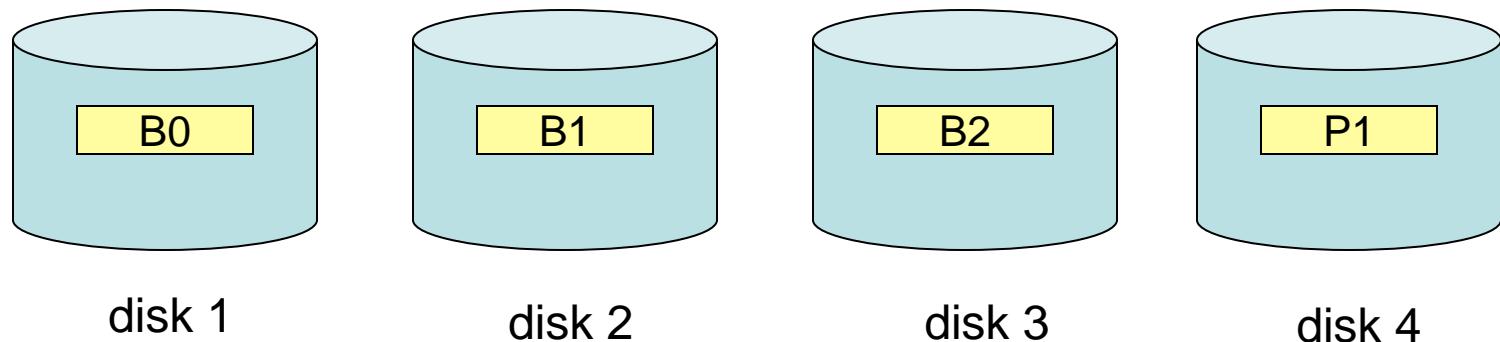
disk 4
mirror of disk 3

RAID levels 2 and 3

- bit level striping (next bit on a separate disk)
- error detection and correction
- RAID 2
 - ECC **error correction** codes (Hamming code)
 - Bit level striping, **several parity bits**
- RAID 3
 - Byte level striping, **single parity bit**
 - error detection by disk controllers (hardware)
- RAID 4
 - Block level striping, single parity bit

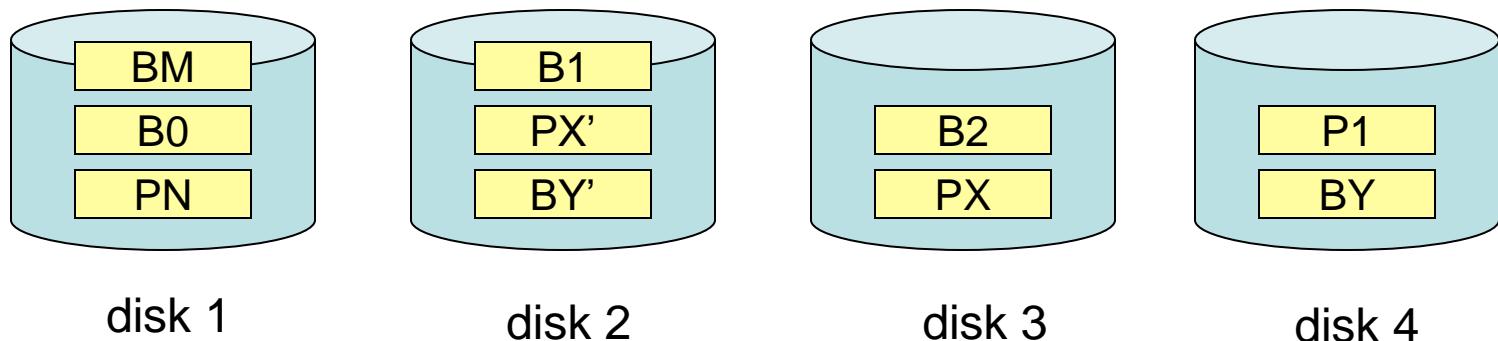
RAID level 4

- block level striping
- parity block for each block in data disks
 - $P1 = B0 \text{ XOR } B1 \text{ XOR } B2$
 - $B2 = B0 \text{ XOR } B1 \text{ XOR } P1$
- an update:
 - $P1' = B0' \text{ XOR } B0 \text{ XOR } P1$ (every update -> **must write parity disk**)



RAID level 5 and 6

- subsumes RAID 4
- parity disk not a bottleneck
 - parity blocks distributed on all disks
- RAID 6
 - tolerates two disk failures
 - P+Q redundancy scheme
 - 2 bits of redundant data for each 4 bits of data
 - more expensive writes



What pages contain logically?

- Files:
 - Physically partitioned into **pages** (or blocks)
 - Logically partitioned into **records**
- Each file is a sequence of records
- Each record is a sequence of fields

student	
<u>cid</u>	name
00112233	Paul

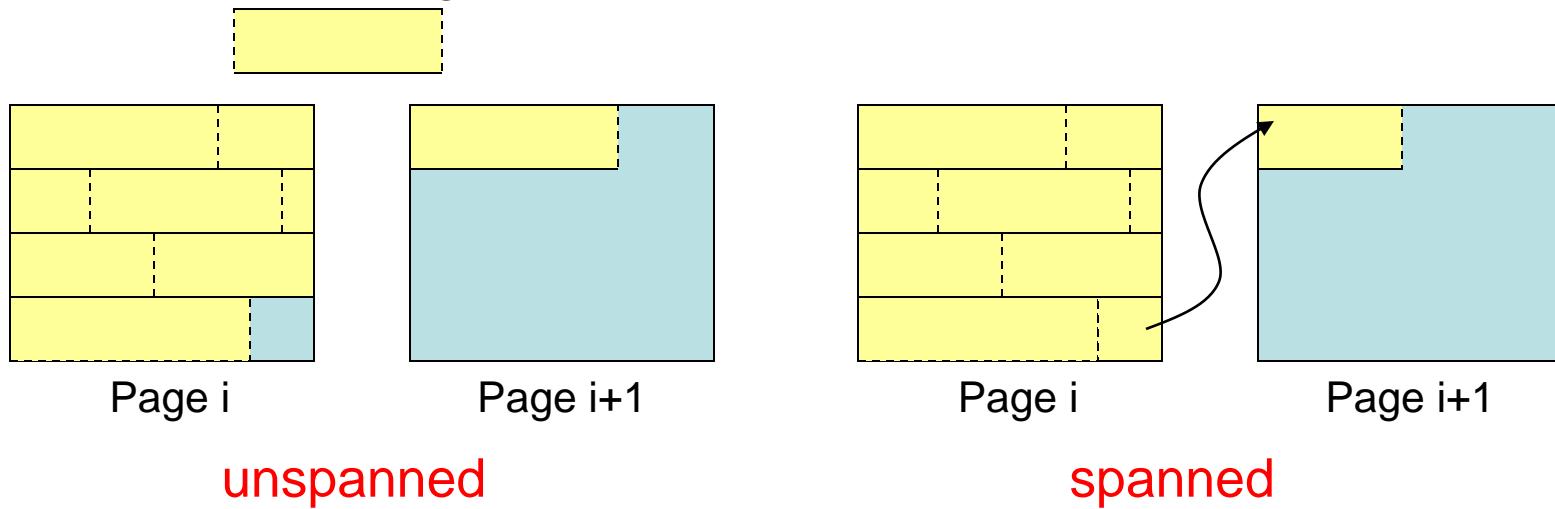
student record:

00112233	Paul
----------	------

$$8 + 4 = 12 \text{ Bytes}$$

Page Organization

- Student record size: 12 Bytes
- Typical page size: 2 KB
- Record identifiers: <Page identifier, offset>
- How records are distributed into pages:
 - Unspanned organization
 - Blocking factor = $\left\lfloor \frac{\text{pagesize}}{\text{recordsize}} \right\rfloor$
 - Spanned organization



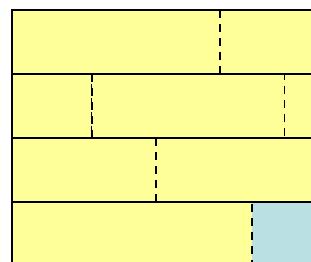
Blocking

- **Blocking:**
 - Refers to storing a number of records in one block on the disk.
- **Blocking factor (bf)** refers to the number of records per block.
- There may be empty space in a block if an integral number of records do not fit in one block.
- File records can be **unspanned** or **spanned**
 - **Unspanned**: no record can span two blocks
 - **Spanned**: a record can be stored in more than one block

The physical disk blocks that are allocated to hold the records of a file can be *contiguous, linked, or indexed*.

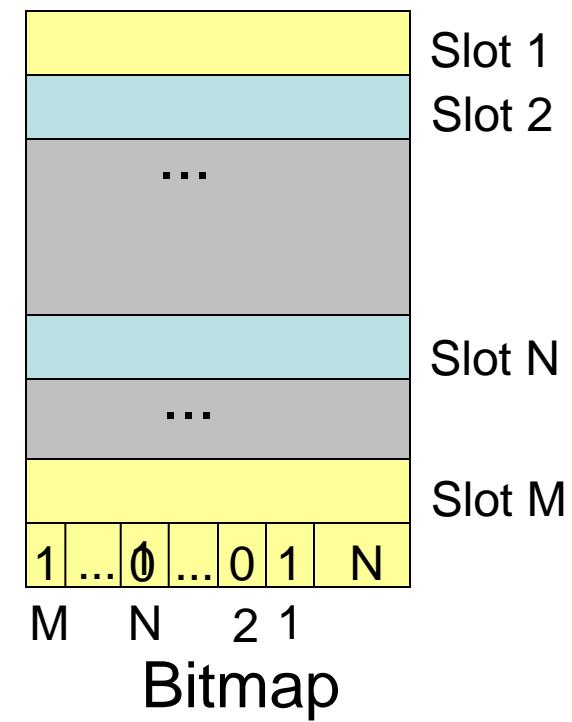
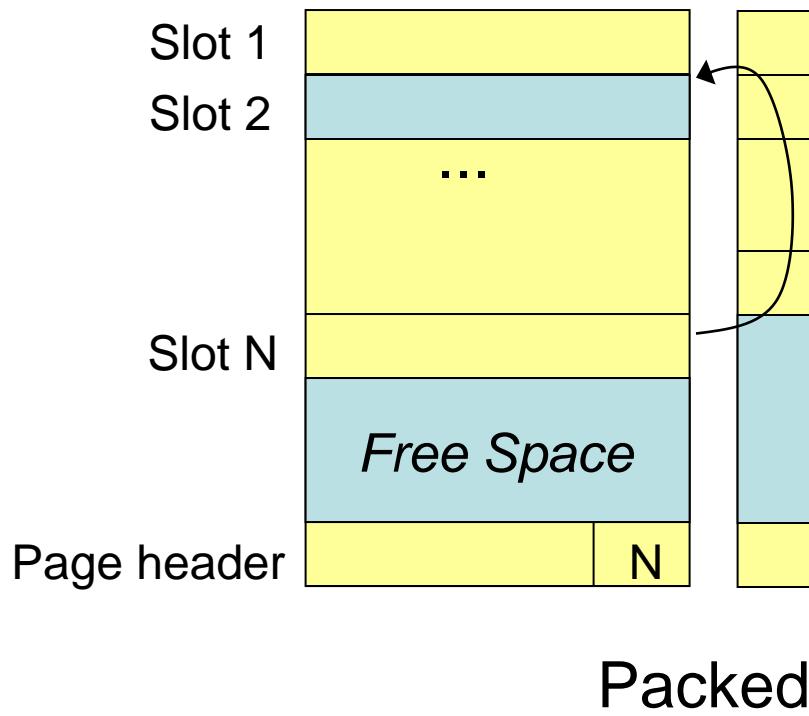
What if a record is deleted?

- Depending on the type of records:
 - Fixed-length records
 - Variable-length records



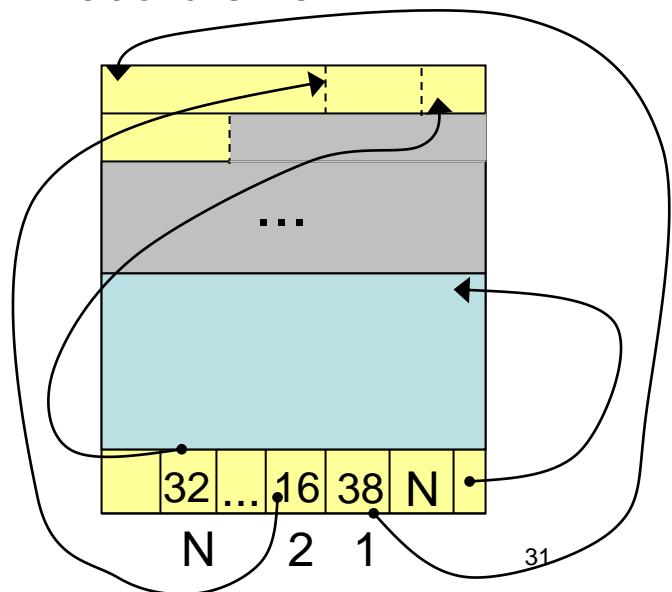
Fixed-length record files

- Upon record deletion:
 - Packed page scheme
 - Bitmap



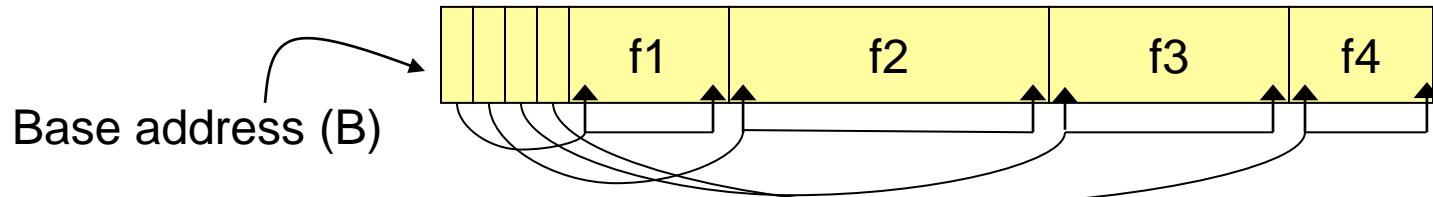
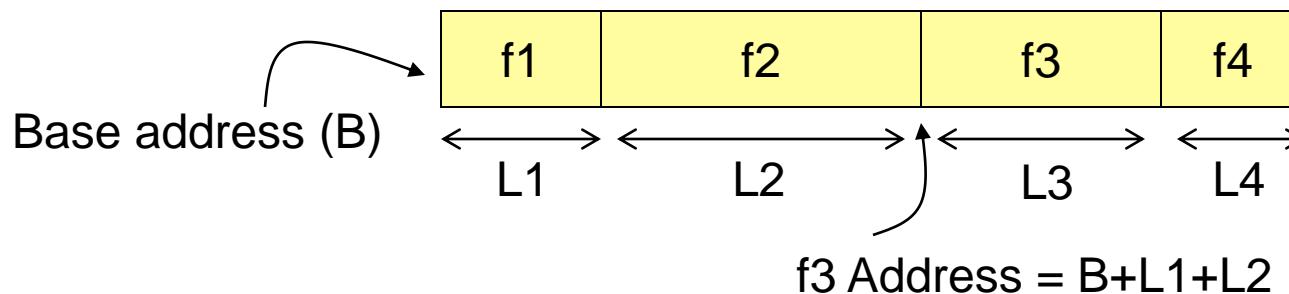
Variable-length record files

- When do we have a file with variable-length records?
 - Column datatype: variable length
 - create table t (field1 int, field2 **varchar2(n)**)
- Problems:
 - Holes created upon deletion have variable size
 - Find large enough free space for new record
- Could use previous approaches: maximum record size
 - a lot of space wasted
- **Use slotted page structure**
 - Slot directory
 - Each slot storing offset, size of record
 - Record IDs: page number, slot number



Record Organization

- Fixed-length record formats
 - Fields stored consecutively
- Variable-length record formats
 - Array of offsets
 - NULL values when start offset = end offset



Operation on Files

- Typical file operations include:
 - **OPEN**: Prepares the file for access and associates a pointer that will refer to a *current* file record at each point in time.
 - **FIND**: Searches for the first file record that satisfies a certain condition and makes it the current file record.
 - **FINDNEXT**: Searches for the next file record (from the current record) that satisfies a certain condition and makes it the current file record.
 - **READ**: Reads the current file record into a program variable.
 - **INSERT**: Inserts a new record into the file & makes it the current file record.
 - **DELETE**: Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid.
 - **MODIFY**: Changes the values of some fields of the current file record.
 - **CLOSE**: Terminates access to the file.
 - **REORGANIZE**: Reorganizes the file records.
 - For example, the records marked deleted are physically removed from the file or a new organization of the file records is created.
 - **READ_ORDERED**: Read the file blocks in order of a specific field of the file.

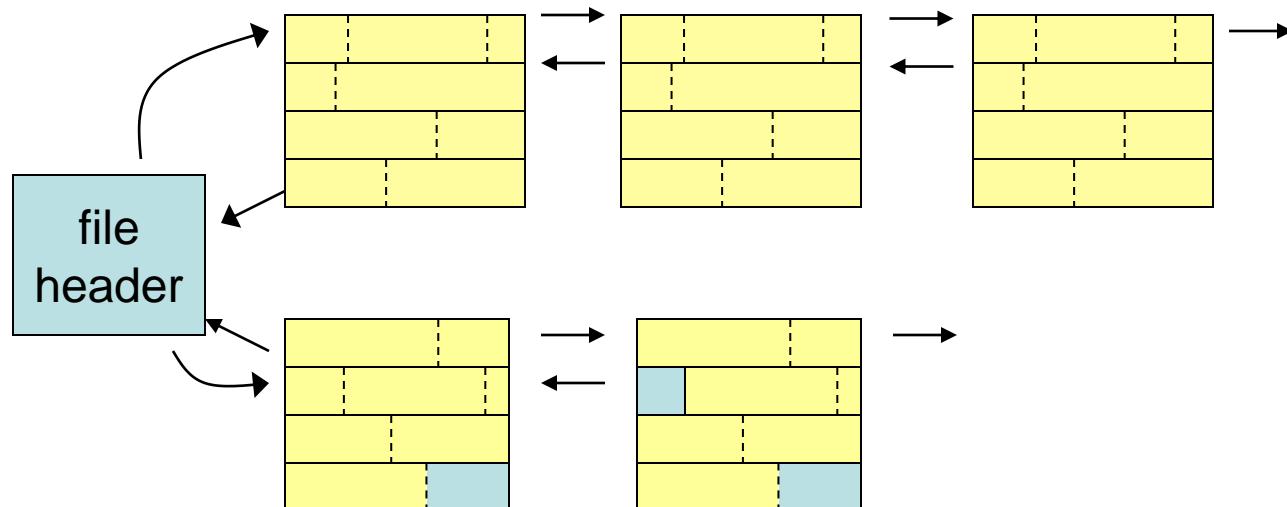
File Organization

(later we study it in a more detailed way)

- Heap files: unordered records
- Sorted files: ordered records
- Hashed files: records partitioned into buckets

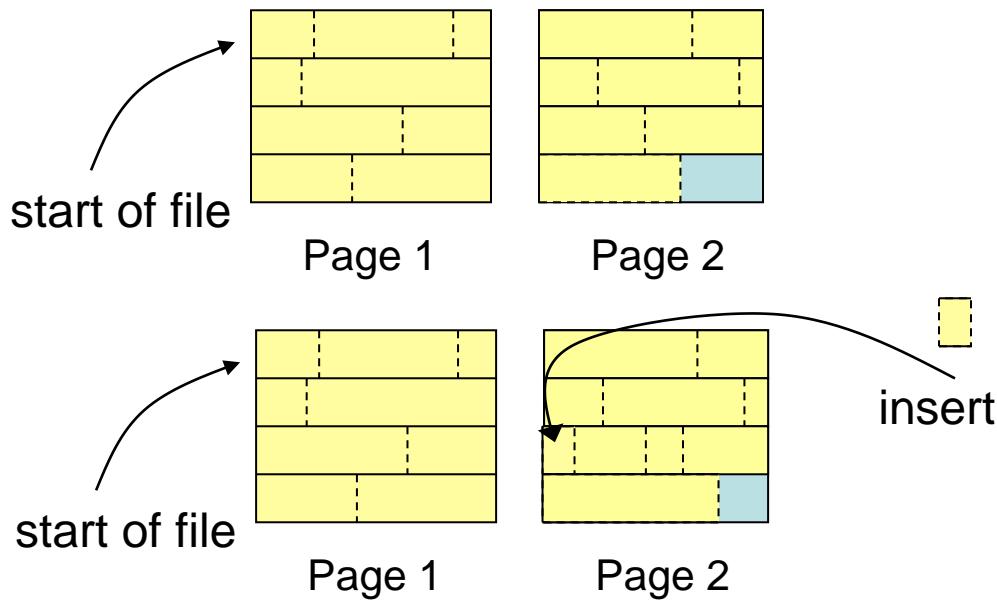
Heap Files

- Simplest file structure
- Efficient insert
- **Slow search and delete**
 - Equality search: half pages fetched on average
 - Range search: all pages must be fetched



Sorted (Ordered) files

- Sorted records based on **ordering field**
 - If ordering field same as key field, **ordering key field**
- **Slow inserts** and deletes
- **Fast logarithmic search**

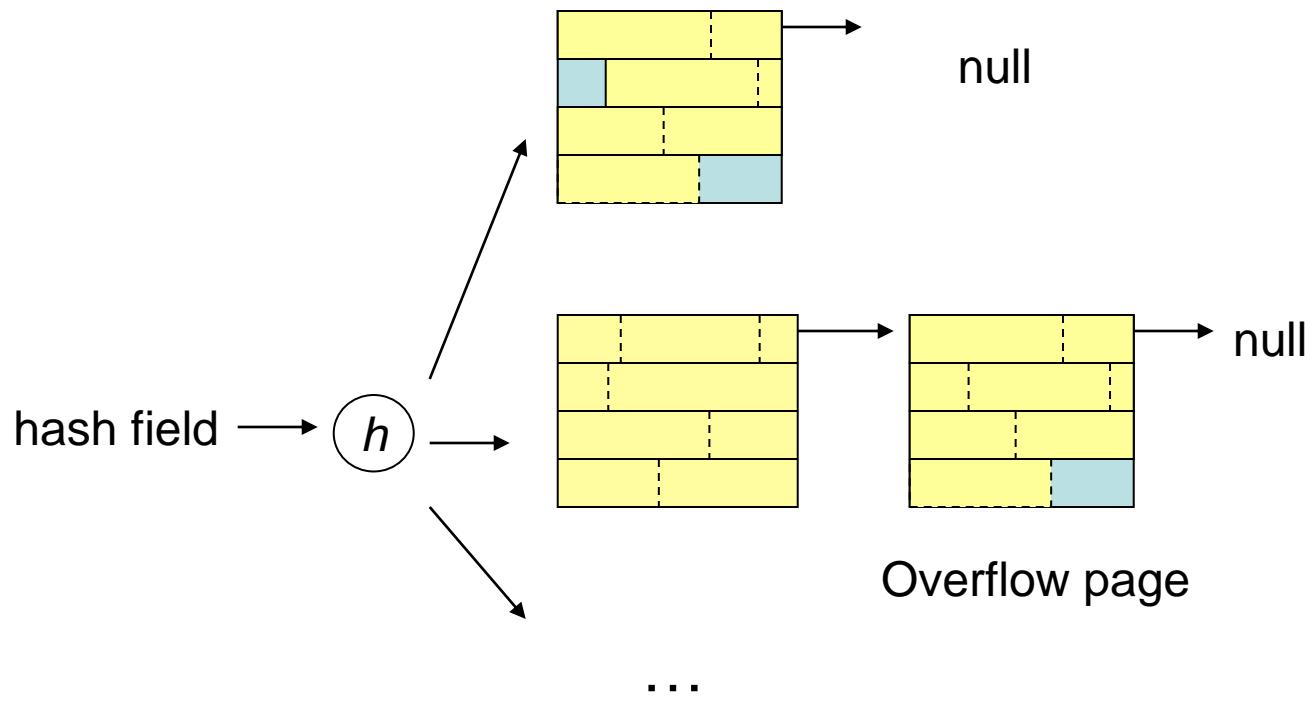


Sorted (Ordered) Files

- Also called a **sequential file**.
- File records are kept sorted by the values of an *ordering field*.
- Insertion is expensive: records must be inserted in the correct order.
 - It is common to keep a separate **unordered overflow** (or *transaction*) **file** for new records to improve insertion efficiency; this is periodically merged with the main ordered file.
- A **binary search** can be used to search for a record on its *ordering field* value.
 - This requires reading and searching \log_2 of the file blocks on the average, an improvement over linear search.
- Reading the records in order of the ordering field is quite efficient.

Hashed Files

- Hash function h on **hash field** distributes pages into buckets
- Efficient equality searches, inserts and deletes
- No support for range searches



Hashed Files

- Hashing for disk files is called **External Hashing**
- The file blocks are divided into M equal-sized **buckets**, numbered $\text{bucket}_0, \text{bucket}_1, \dots, \text{bucket}_{M-1}$.
 - Typically, a bucket corresponds to one (or a fixed number of) disk block.
- One of the file fields is designated to be the **hash key** of the file.
- The record with hash key value K is stored in bucket i , where $i=h(K)$, and h is the **hashing function**.
- Search is very efficient on the hash key.
- Collisions occur when a new record hashes to a bucket that is already full.
 - An **overflow file** is kept for storing such records.
 - Overflow records that hash to each bucket can be linked together.

Hashed Files

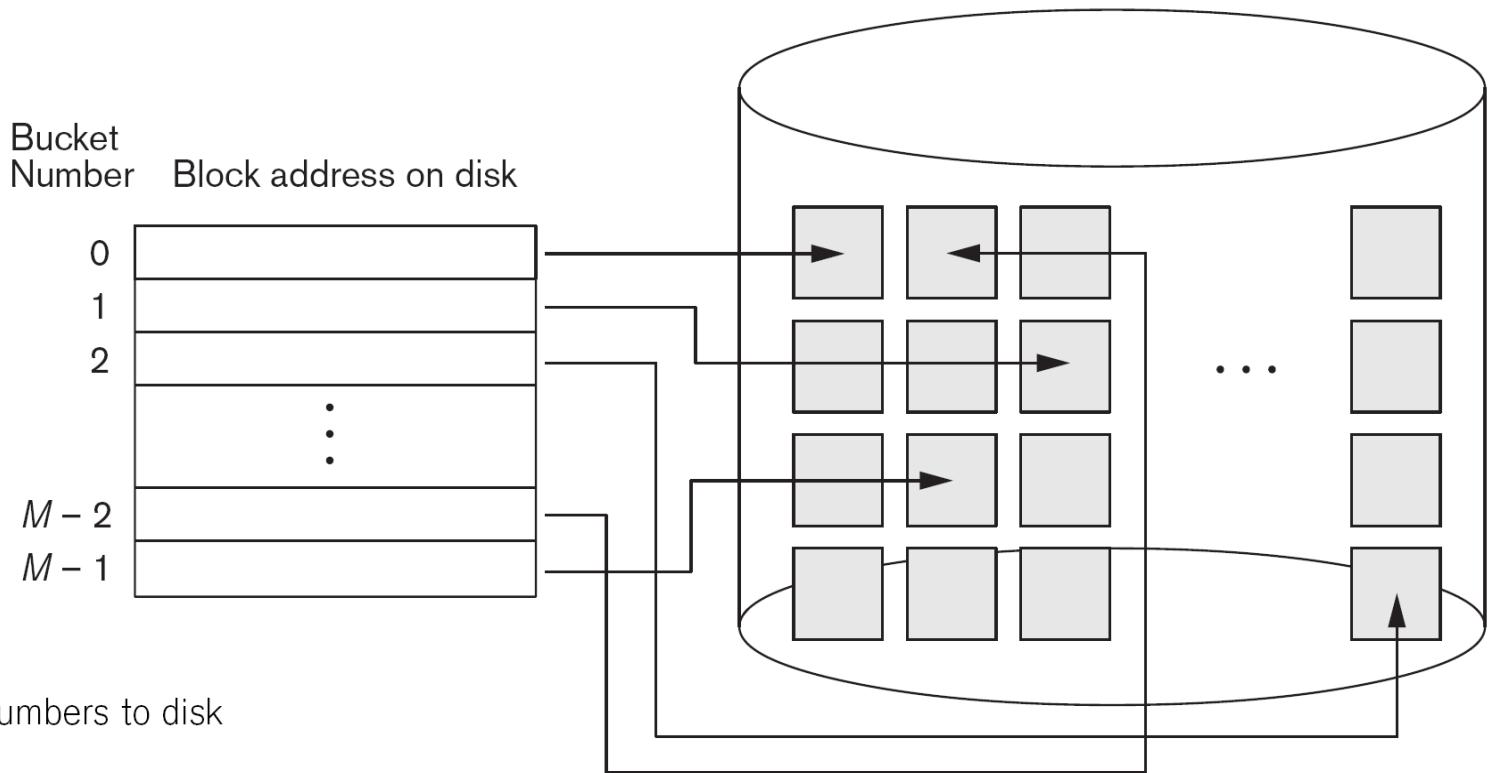


Figure 17.9

Matching bucket numbers to disk block addresses.

Summary (1/2)

- Why Physical Storage Organization?
 - understanding low-level details which affect data access
 - make data access more efficient
- Primary Storage, Secondary Storage
 - memory fast
 - disk slow but non-volatile
- Data stored in files
 - partitioned into pages physically
 - partitioned into records logically
- Optimize I/Os
 - scheduling algorithms
 - RAID
 - page replacement strategies

Summary (2/2)

- File Organization
 - how each file type performs
- Page Organization
 - strategies for record deletion
- Record Organization

Ullman et al. :
Database System Principles

Disk Organization

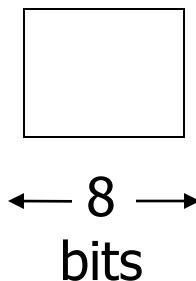
Topics for today

- How to lay out **data** on disk
- How to **move** it to memory

What are the data items we want to store?

- a salary
- a name
- a date
- a picture

➡ What we have available: Bytes



To represent:

- **Integer** (short): 2 bytes
e.g., 35 is

00000000

00100011

- **Real**, floating point
 n bits for mantissa, m for exponent....

To represent:

- **Characters**

→ various coding schemes suggested,
most popular is ascii

Example:

A: 1000001

a: 1100001

5: 0110101

LF: 0001010

To represent:

- Boolean

e.g., TRUE

1111 1111

FALSE

0000 0000

- Application **specific**

e.g., RED → 1 GREEN → 3

BLUE → 2 YELLOW → 4 ...

➡ Can we use less than 1 byte/code?

Yes, but only if desperate...

To represent:

- **Dates**

- e.g.: - Integer, # **days since Jan 1, 1900**
 - 8 characters, YYYYMMDD
 - 7 characters, YYYYDDD
(not YYMMDD! Why?)

- **Time**

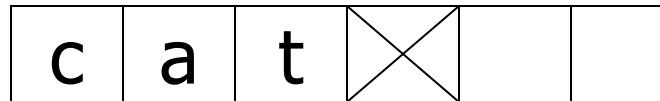
- e.g. - Integer, **seconds since midnight**
 - characters, HHMMSSFF

To represent:

- **String** of characters

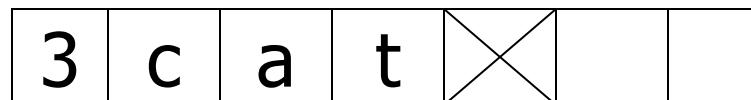
- Null terminated

e.g.,



- Length given

e.g.,



- Fixed length

To represent:

- Bag of bits

Length	Bits
--------	------

Key Point

- 
- **Fixed length** items
 - **Variable length** items
 - usually length given at beginning

Also

- Type of an item: Tells us how to interpret
(plus size if fixed)

Overview

Data Items



Records



Blocks



Files



Memory

Record - Collection of related data items (called **FIELDS**)

E.g.: Employee record:

- name field,
- salary field,
- date-of-hire field, ...

Types of records:

- Main choices:
 - FIXED vs VARIABLE **FORMAT**
 - FIXED vs VARIABLE **LENGTH**

Fixed format

A SCHEMA (of a table record) contains following information

- # fields
- type of each field
- order in record
- meaning of each field

Example: fixed format and length

Employee record

- (1) E#, 2 byte integer
- (2) E.name, 10 char.
- (3) Dept, 2 byte code



Schema

55	s m i t h	02
----	-----------	----



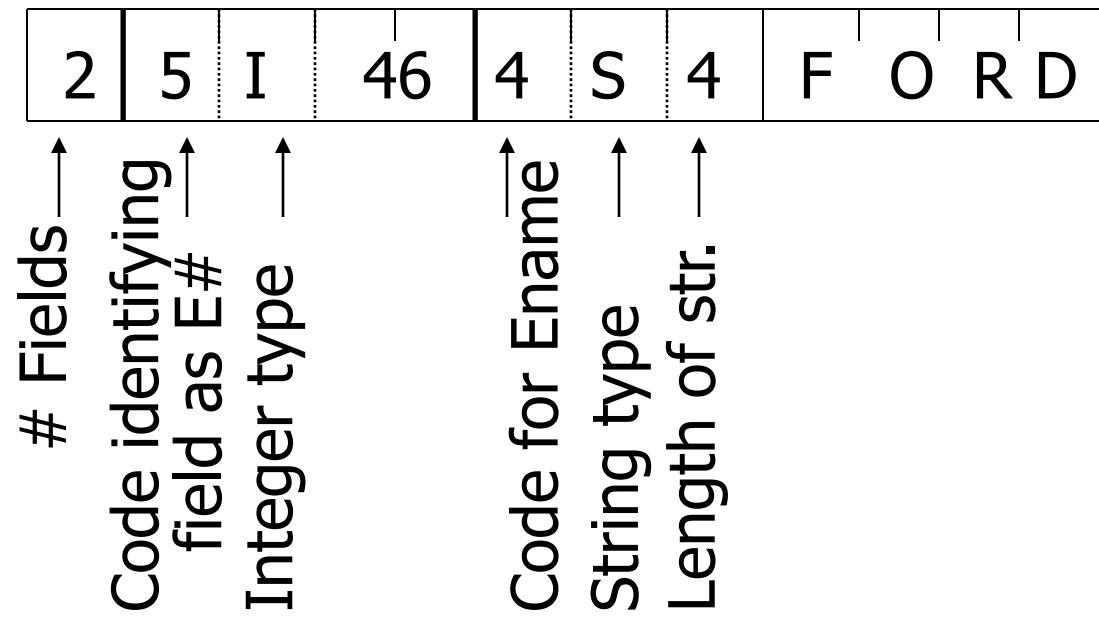
Records

83	j o n e s	01
----	-----------	----

Variable format

- Record itself contains format
“Self Describing”

Example: variable format and length



Field name codes could also be strings, i.e. TAGS

Variable format useful for:

- “sparse” records
- repeating fields
- evolving formats

.....

But may waste space...

- EXAMPLE: var format record with repeating fields

Employee → one or more → children

3	E_name: Fred	Child: Sally	Child: Tom
---	--------------	--------------	------------

Note: Repeating fields does not imply

- variable format, nor
- variable size

John	Sailing	Chess	--
------	---------	-------	----

- Key is to allocate maximum number of repeating fields (if not used → null)

★ Many variants between
fixed - variable format:

Example: Include record type in record

5	27
---	----	------

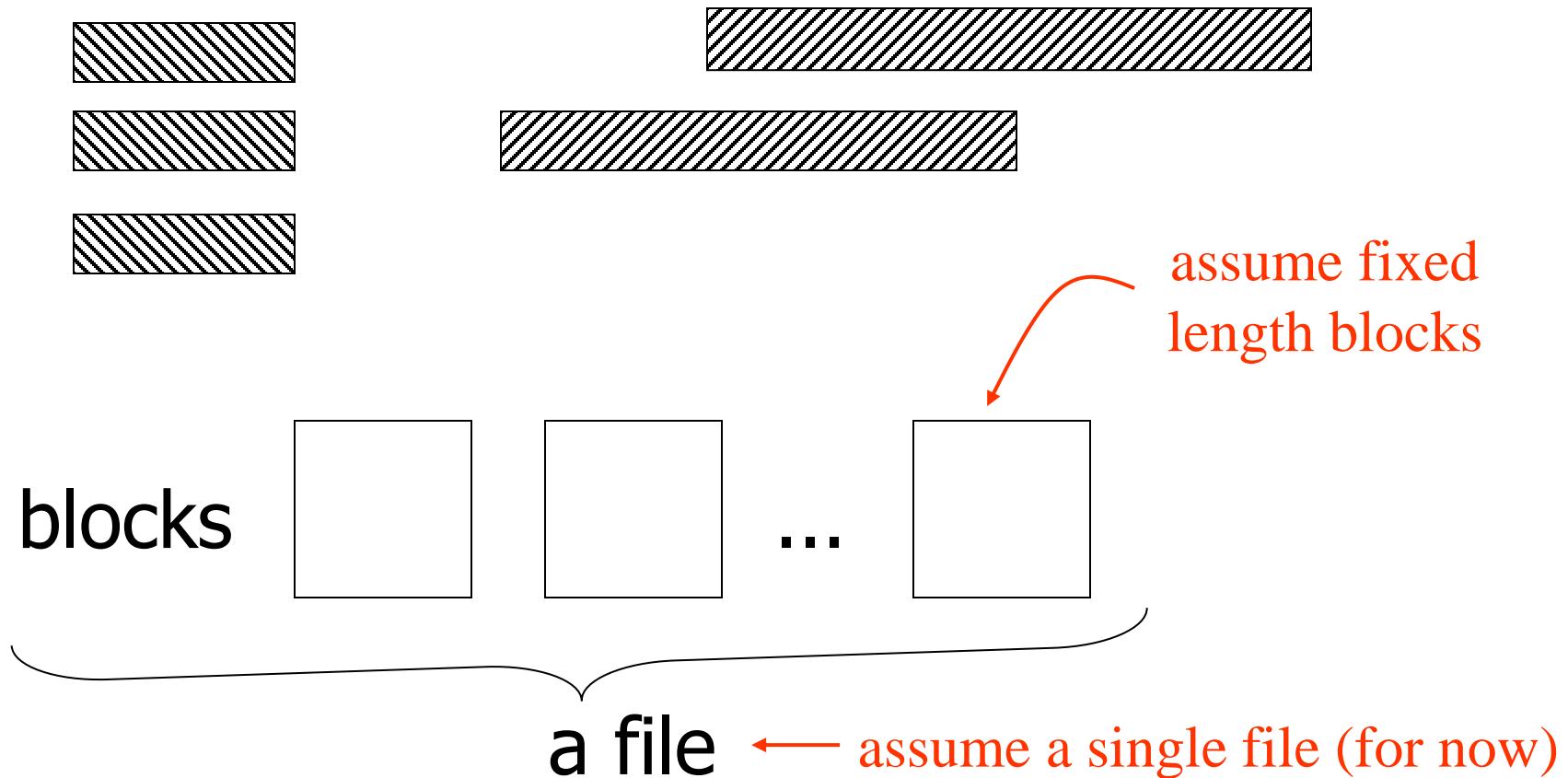
record type record length
tells me what
to expect
(i.e. points to schema)

Record header - data at beginning
that describes record

May contain:

- record type
- record length
- time stamp
- other stuff ...

Next: placing records into blocks



Options for storing **records in blocks**:

- (1) separating records
- (2) spanned vs. unspanned
- (3) sequencing
- (4) indirection

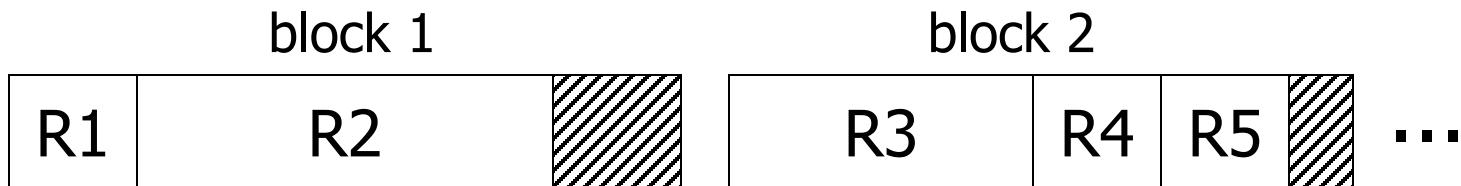
(1) Separating records



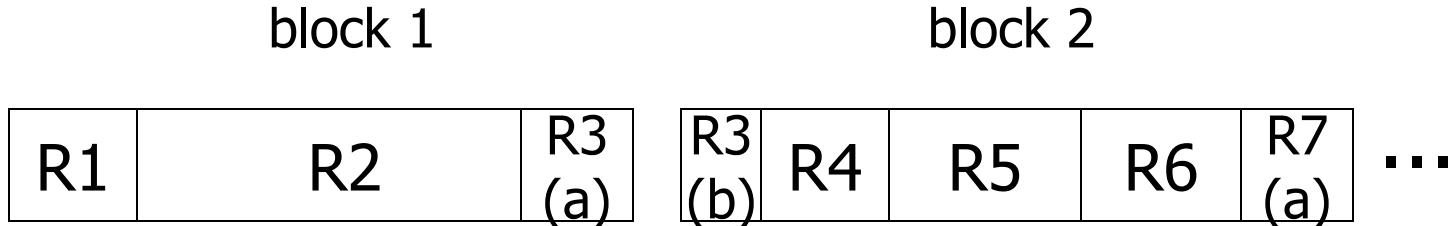
- (a) no need to separate - fixed size recs.
- (b) special marker
- (c) give **record lengths** (or **offsets**)
 - within each record
 - in block header

(2) Spanned vs. Unspanned

- Unspanned: records must be within one block



- Spanned



With spanned records:

R1	R2	R3 (a)	R3 (b)	R4	R5	R6	R7 (a)
----	----	-----------	-----------	----	----	----	-----------

need indication
of partial record
“pointer” to rest

need indication
of continuation
(+ from where?)

Spanned vs. unspanned:

- Unspanned is much simpler, but may waste space...
- Spanned essential if
record size > block size

(3) Sequencing

- Ordering records in file (and block) by some key value

Sequential file (\Rightarrow sequenced)

Why sequencing?

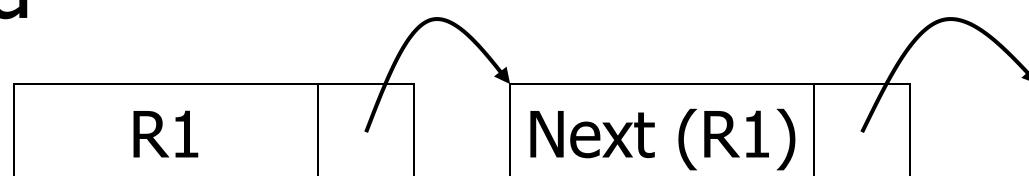
Typically to make it possible to efficiently
read records in order
(e.g., to do a merge-join — discussed later)

Sequencing Options

(a) Next record physically contiguous



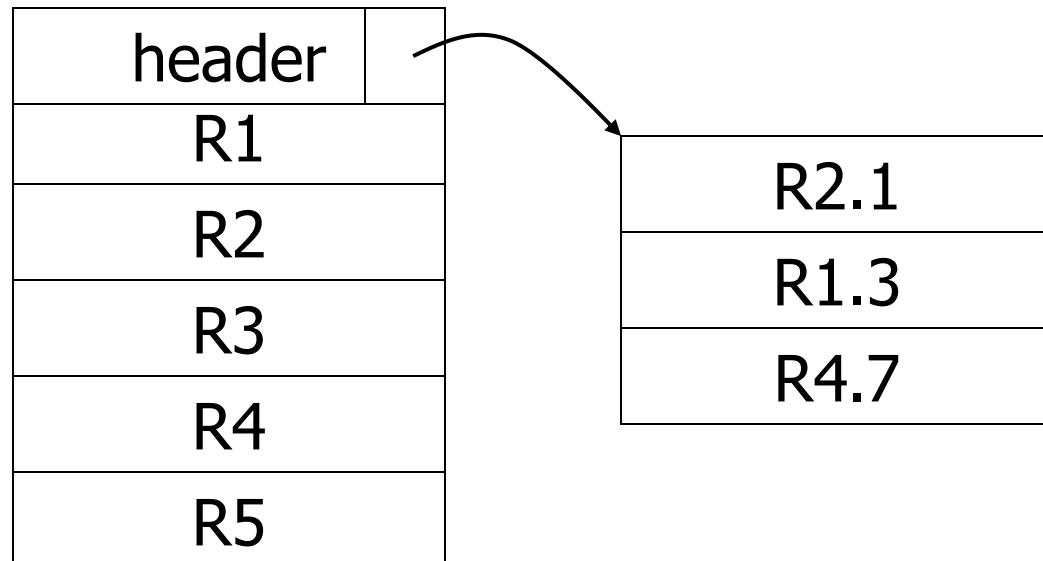
(b) Linked



Sequencing Options

(c) Overflow area

Records
in sequence



(4) Indirection

- How does one refer to records?



(4) Indirection

- How does one refer to records?



Many options:

Physical



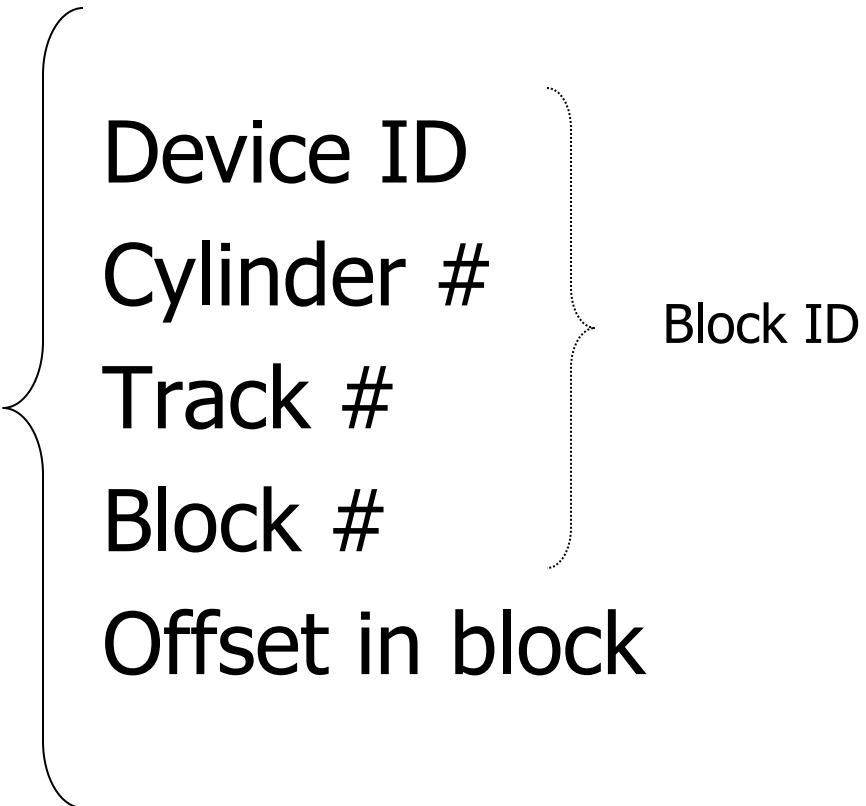
Indirect

★ Purely Physical

E.g., Record
Address = or ID

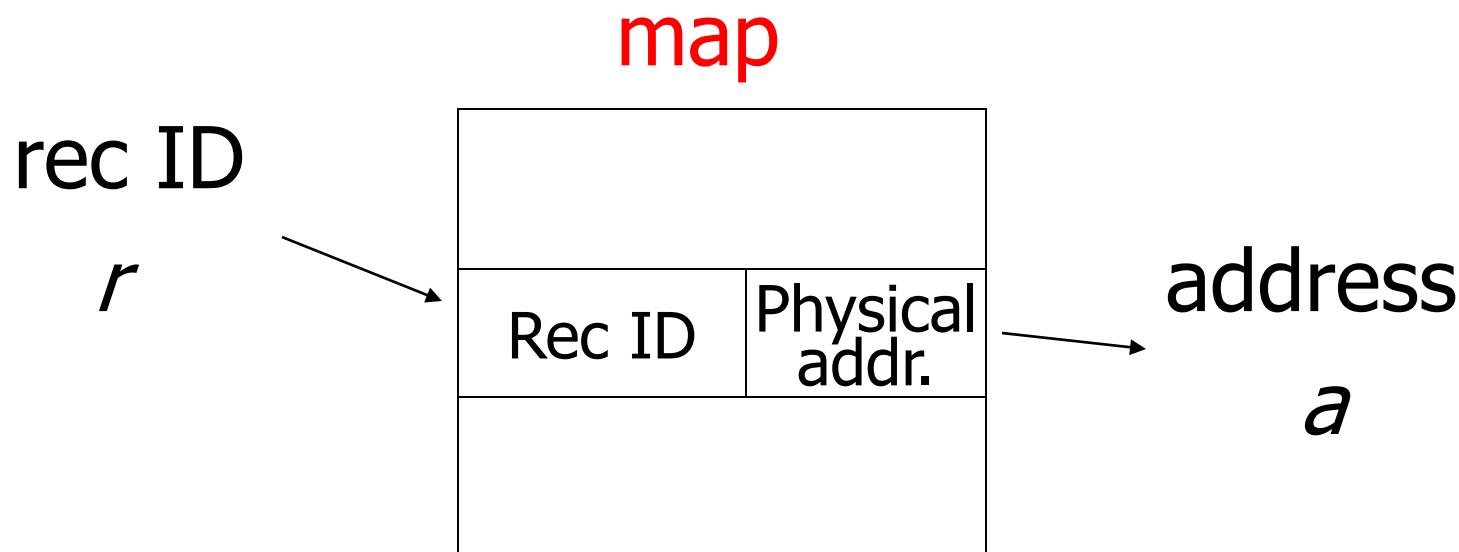
Device ID
Cylinder #
Track #
Block #
Offset in block

Block ID



★ Fully Indirect

E.g., Record ID is arbitrary bit string



Tradeoff

Flexibility \longleftrightarrow

to move records

(for deletions, insertions)

Cost

of indirection

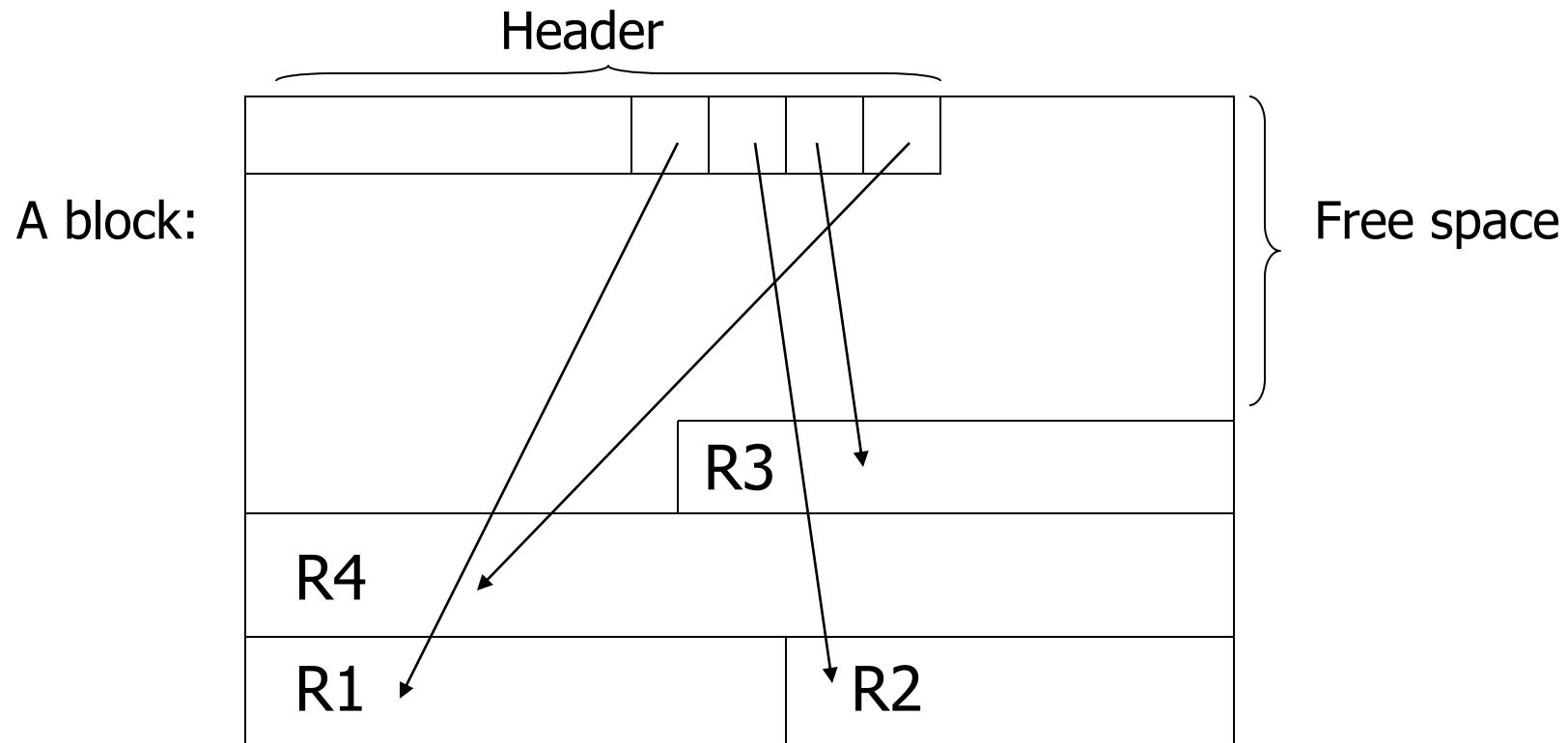
(manage the map)

Physical \longleftrightarrow Indirect



Many options
in between ...

Example: Indirection in block



Block header - data at beginning that describes block

May contain:

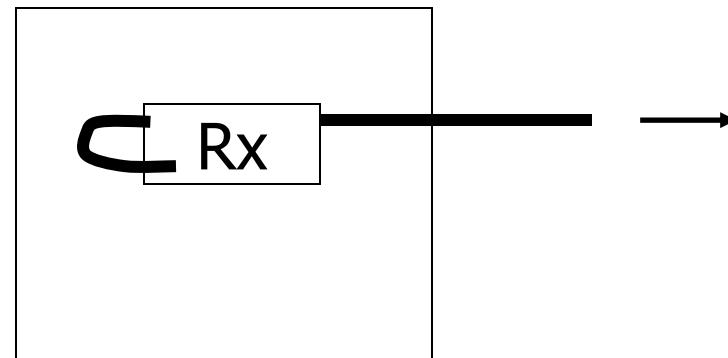
- File ID (or RELATION or DB ID)
- This block ID
- Record directory
- Pointer to free space
- Type of block (e.g. contains recs type 4;
is overflow, ...)
- Pointer to other blocks “like it”
- Timestamp ...

Other Topics

- (1) Insertion/Deletion
- (2) Buffer Management
- (3) Comparison of Schemes

Deletion

Block



Options:

- (a) Immediately reclaim space
- (b) Mark deleted
 - May need chain of deleted records (for re-use)
 - Need a way to mark:
 - special characters
 - delete field
 - in map

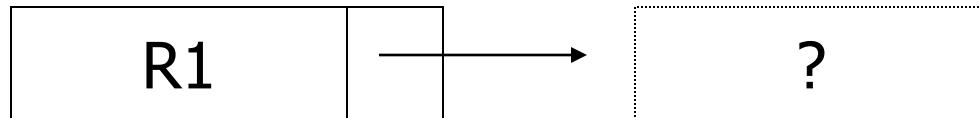


As usual, many tradeoffs...

- How expensive is to move valid record to free space for immediate reclaim?
- How much space is wasted?
 - e.g., deleted records, delete fields, free space chains,...

Concern with deletions

Dangling pointers



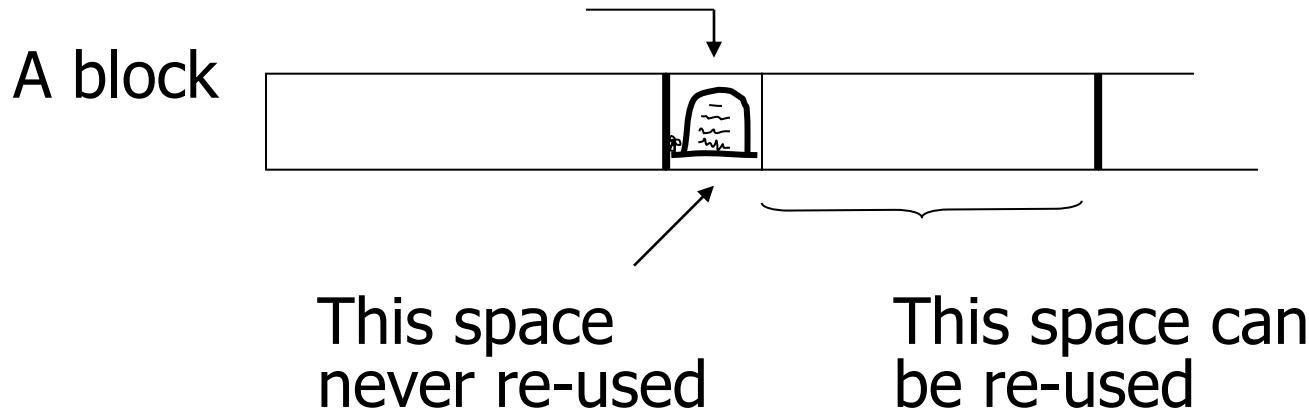
Solution #1: Do not worry

We can **never reuse the space** of the deleted record.

Solution #2: Tombstones

E.g., Leave “MARK” in map or old location

- Physical IDs



Solution #2: Tombstones

E.g., Leave “MARK” **in map** or old location

- Logical IDs

map

ID	LOC
7788	



Never reuse
ID 7788 nor
space in map...

Insert

Easy case: records not in sequence

- Insert new record at end of file or in deleted slot
- If records are variable size, not as easy...

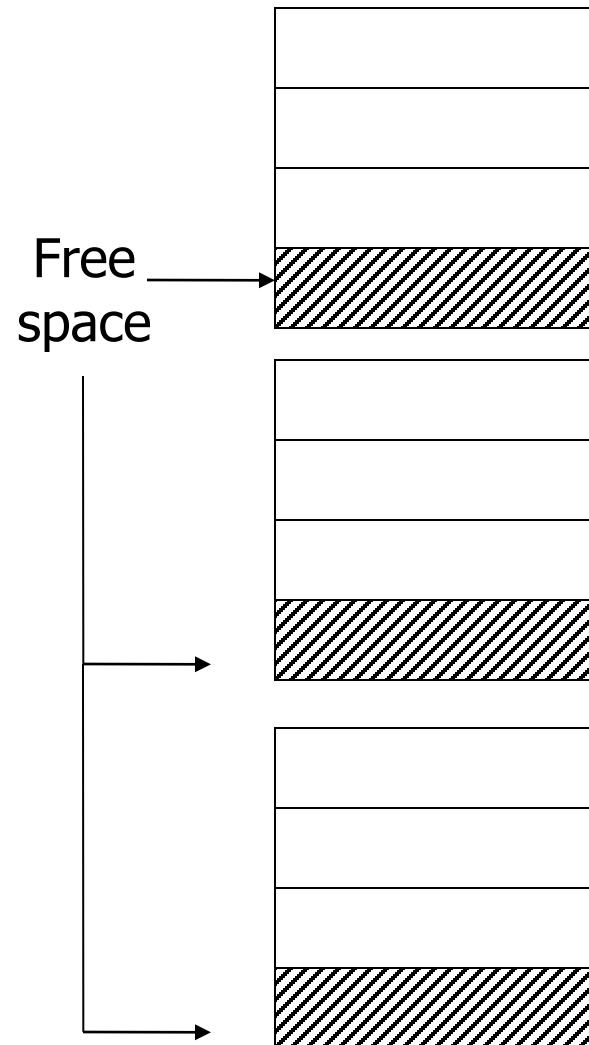
Insert

Hard case: records in sequence

- If free space “close by”, not too bad...
- Or use overflow idea...

Interesting problems:

- How much free space to leave in each block, track, cylinder?
- How often do I reorganize file + overflow?



Buffer Management

- DB features needed
- Why LRU may be bad
- Pinned blocks
- Forced output
- Double buffering

Row vs Column Store

- So far, we assumed that fields of a record are stored contiguously (row store)...
- Another option is to store like fields together (column store)

Row Store

- Example: Order consists of
 - id, cust, prod, store, price, date, qty

id1	cust1	prod1	store1	price1	date1	qty1
-----	-------	-------	--------	--------	-------	------

id2	cust2	prod2	store2	price2	date2	qty2
-----	-------	-------	--------	--------	-------	------

id3	cust3	prod3	store3	price3	date3	qty3
-----	-------	-------	--------	--------	-------	------

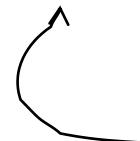
Column Store

- Example: Order consists of
 - id, cust, prod, store, price, date, qty

id1	cust1
id2	cust2
id3	cust3
id4	cust4
...	...

id1	prod1
id2	prod2
id3	prod3
id4	prod4
...	...

id1	price1	qty1
id2	price2	qty2
id3	price3	qty3
id4	price4	qty4
...



ids may or may not be stored explicitly

Row vs Column Store

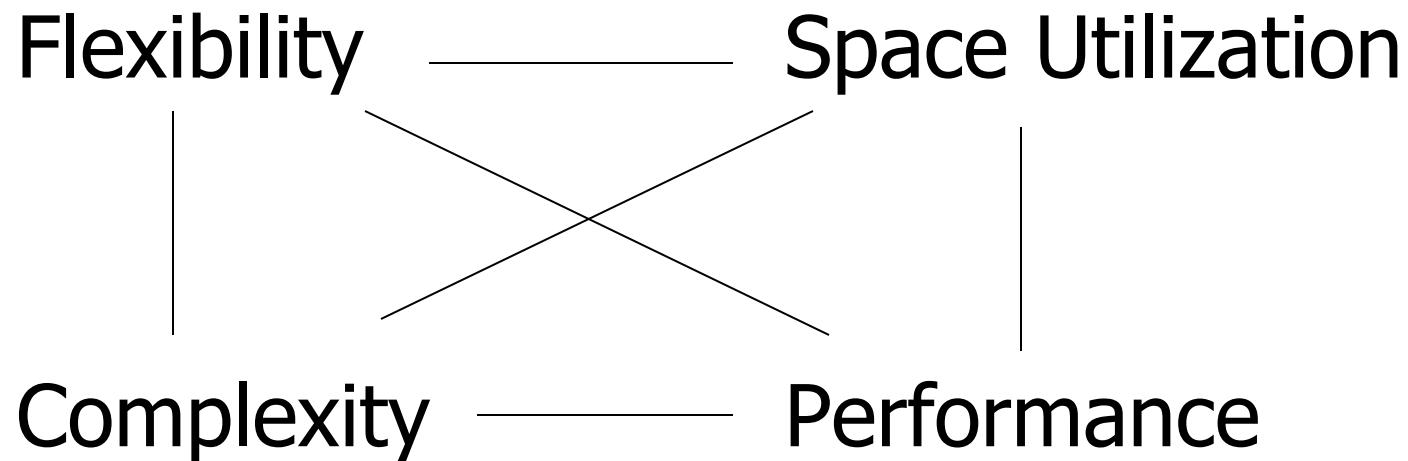
- Advantages of Column Store
 - more compact storage (fields need not start at byte boundaries)
 - efficient reads on data mining operations
- Advantages of Row Store
 - writes (multiple fields of one record) more efficient
 - efficient reads for record access (OLTP)

Comparison

- There are 10,000,000 ways to organize my data on disk...

Which is right for me?

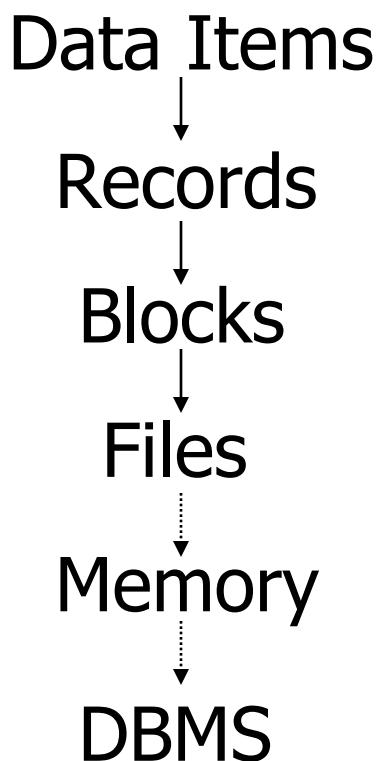
Issues:



- ★ To evaluate a given strategy, compute following parameters:
 - > space used for expected data
 - > expected time to
 - fetch record given key
 - fetch record with next key
 - insert record
 - append record
 - delete record
 - update record
 - read all file
 - reorganize file

Summary

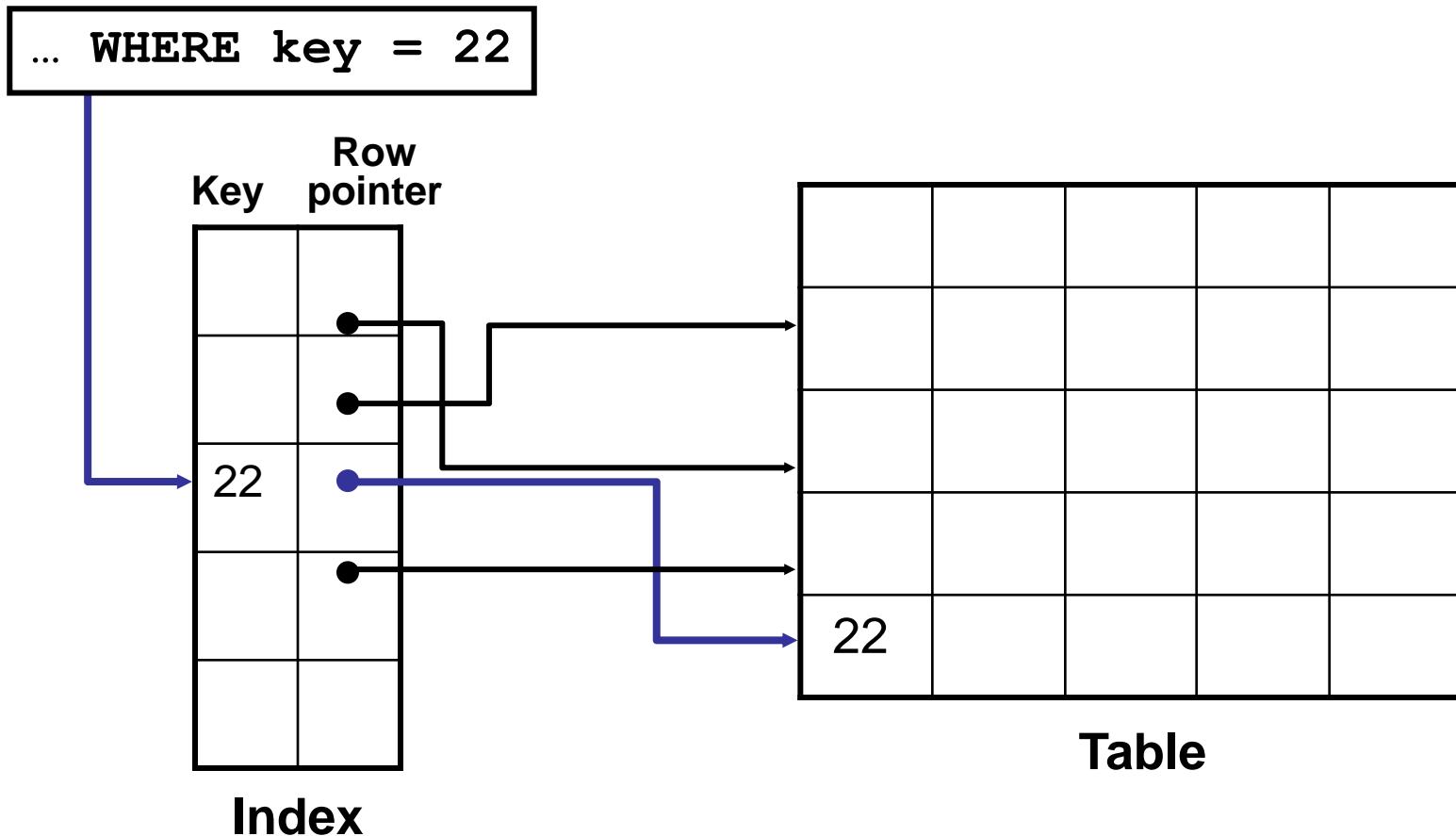
- How to lay out data on disk



Next

How to find a record quickly,
given a key

Indexes

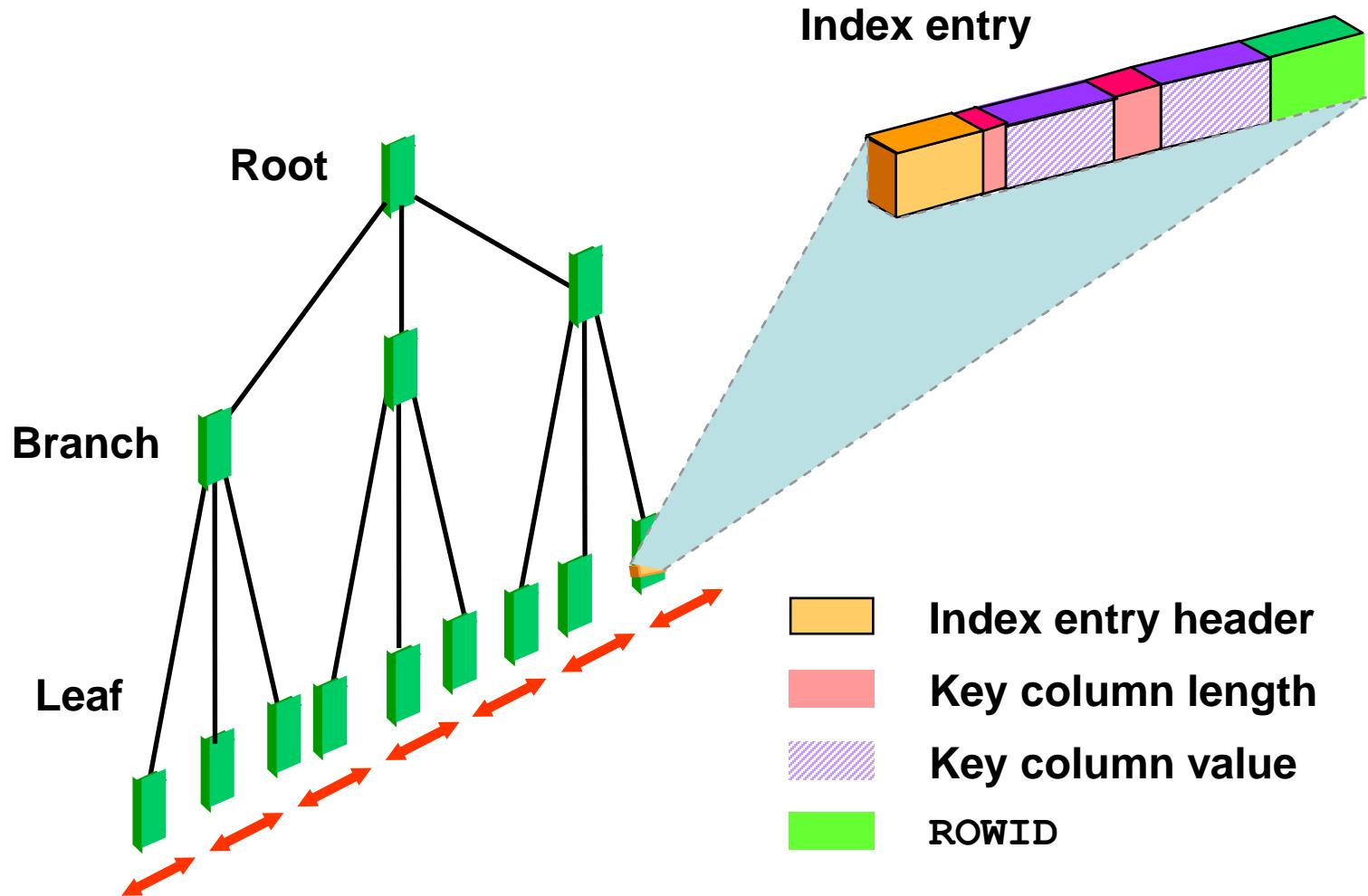


Types of Indexes

These are several types of index structures available to you, depending on the need:

- A **B+-tree** index is in the form of a balanced tree and is the default index type.
- A **bitmap** index has a bitmap for each distinct value indexed, and each bit position represents a row that may or may not contain the indexed value. This is best for low-cardinality columns.

B+ tree index



B+-Tree Index

Structure of a B+-tree index

At the top of the index is the root, which contains entries that point to the next level in the index. At the next level are branch blocks, which in turn point to blocks at the next level in the index. At the lowest level are the leaf nodes, which contain the index entries that point to rows in the table. The leaf blocks are doubly linked to facilitate the scanning of the index in an ascending as well as descending order of key values.

Format of index leaf entries

An index entry is made up of the following components:

An entry header, which stores the number of columns and locking information

Key column length-value pairs, which define the size of a column in the key followed by the value for the column (The number of such pairs is a maximum of the number of columns in the index.)

ROWID of a row that contains the key values

Index Options

- A unique index ensures that every indexed value is unique.

CREATE **UNIQUE** INDEX emp1 ON EMP (ename);

DBA_INDEXES.UNIQUENESS → 'UNIQUE'

- Bitmap index

CREATE **BITMAP** INDEX emp2 ON EMP (deptno);

DBA_INDEXES.INDEX_TYPE → 'BITMAP'

An index can have its key values stored in ascending or descending order.

CREATE INDEX emp3 ON emp (sal **DESC**);

DBA_IND_COLUMNS.DESCEND → 'DESC'

Index Options

- A **composite index** is one that is based on more than one column.

CREATE INDEX emp4 ON emp (empno, sal);

DBA_IND_COLUMNS.COLUMN_POSITION → 1,2 ...

- A **function-based index** is an index based on a function's return value.

CREATE INDEX emp5 ON emp (SUBSTR(ename, 3, 4));

DBA_IND_EXPRESSIONS.COLUMN_EXPRESSION → 'SUBSTR ...'

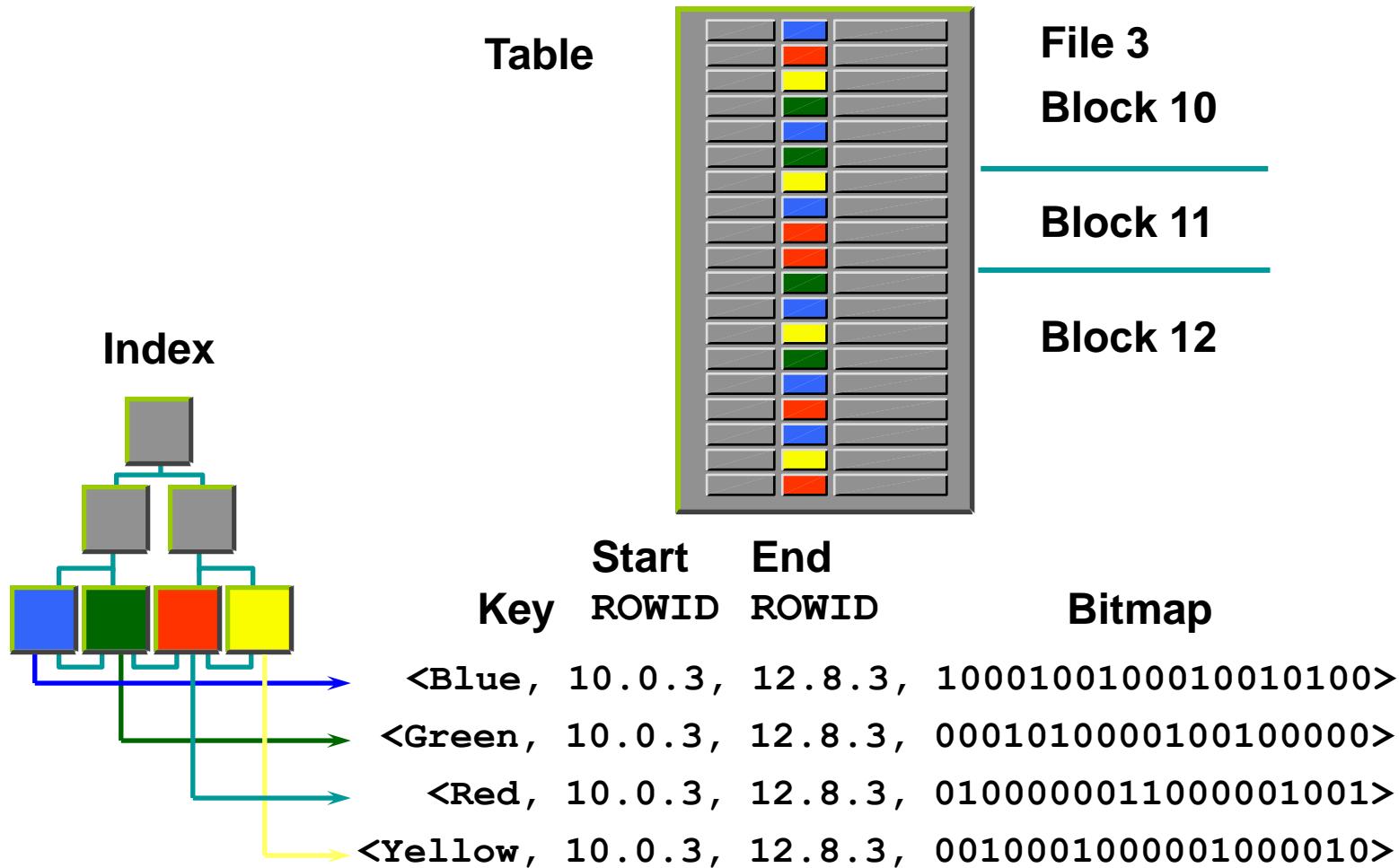
- A **compressed index** has repeated key values removed.

CREATE INDEX emp6 ON emp (empno, ename, sal) COMPRESS 2;

DBA_INDEXES.COMPRESSION → 'ENABLED'

DBA_INDEXES.PREFIX_LENGTH → 2

Bitmap Indexes



Bitmap Indexes

Structure of a bitmap index

A bitmap index is also organized as a B-tree, but the **leaf node stores a bitmap for each key value** instead of a list of ROWIDs. Each bit in the bitmap corresponds to a possible ROWID, and if the bit is set, it means that the row with the corresponding ROWID contains the key value.

As shown in the diagram, the leaf node of a bitmap index contains the following:

- An entry header that contains the number of columns and lock info
- Key values consisting of length and value pairs for each key column
- Start ROWID
- End ROWID

A bitmap segment consisting of a string of bits. (The bit is set when the corresponding row contains the key value and is unset when the row does not contain the key value. The **Oracle** server **uses a patented compression** technique **to store bitmap segments.**)

Bitmap Index

Empno	Status	Region	Gender	Info
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	bracket_3

REGION='east'	REGION='central'	REGION='west'
1	0	0
0	1	0
0	0	1
0	0	1
0	1	0
0	1	0

Using Bitmap Indexes

```
SELECT COUNT(*)  
FROM CUSTOMER  
WHERE MARITAL_STATUS = 'married'  
AND REGION IN ('central','west');
```

status = 'married'	region = 'central'	region = 'west'			
0	0	0	0	0	0
1	1	0	1	1	1
1	AND	0	1	1	1
0	0	1	0	1	0
0	1	0	0	1	0
1	1	0	1	1	1

Range queries

AGE	SALARY
25	60
45	60
50	75
50	100
50	120
70	110
85	140
30	260
25	400
45	350
50	275
60	260

```
SELECT * FROM T  
WHERE Age BETWEEN 44 AND 55  
AND Salary BETWEEN 100 AND 200;
```

Bitvectors for Age

25: 100000001000
30: 000000010000
45: 010000000100
50: 001110000010
60: 000000000001
70: 000001000000
85: 000000100000

Bitvectors for Salary

60: 110000000000
75: 001000000000
100: 000100000000
110: 000001000000
120: 000010000000
140: 000000100000
260: 000000010001
275: 000000000010
350: 000000000100
400: 000000001000

Range queries

AGE	SALARY
25	60
45	60
50	75
50	100
50	120
70	110
85	140
30	260
25	400
45	350
50	275
60	260

```
SELECT * FROM T  
WHERE Age BETWEEN 44 AND 55  
AND Salary BETWEEN 100 AND 200;
```

45: 010000000100

50: 001110000010 OR -> 011110000110

100: 000100000000

110: 000001000000

120: 000010000000

140: 000000100000 OR -> 000111100000

011110000110

000111100000 AND -> 000110000000

Compressed bitmaps

1's in a bit vector will be very rare. We compress the vector.

Run-length encoding:

run: a sequence of i 0's followed by a 1

10000001000000000100010000000000001

1. Determine how many bits the binary representation of i has. This is number j .
2. We represent j in „unary” by $j-1$ 1's and a single 0.
3. Then we follow with i in binary.

Compressed bitmaps

Example: 1000000000000001
run with 13 0's

$j = 4 \rightarrow$ in unary: 1110

i in binary: 1101

Encoding for the run: 11101101

Compressed bitmaps

Encoding for $i = 0$: 00

Encoding for $i = 1$: 01

We ignore the trailing 0's. But not the starting 0's !

Decoding:

Decode the following: 11101101001011 -> 13, 0, 3

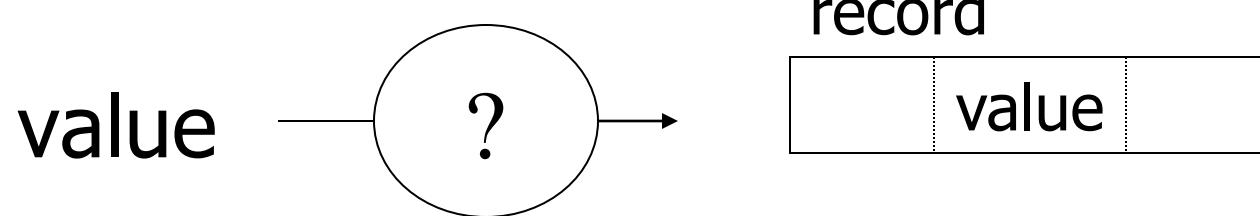
Original bitvector: 0000000000000110001

Ullman et al. :
Database System Principles

Notes 4: Indexing

Chapter 4

Indexing & Hashing



Topics

- Conventional indexes
- B-trees
- Hashing schemes

- A single-level index is an **auxiliary file** that makes it more efficient to search for a record in the data file.
- The index is **usually** specified **on one field** of the file (although it could be specified on several fields).
- One form of an index is a file of entries **<field value, pointer to record>**, which is **ordered by field value**.
- The index is called an **access path** on the field.

- The index file usually occupies considerably less disk blocks than the data file because **its entries are much smaller**
- A binary search on the index yields a pointer to the file record
- Indexes can also be characterized as dense or sparse
 - A **dense index** has an index entry for every search key value (usually every record) in the data file.
 - A **sparse (or nondense) index**, on the other hand, has index entries for only some of the search values (typically one entry per data file block)

Sequential File

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

Dense Index

10	
20	
30	
40	

50	
60	
70	
80	

90	
100	
110	
120	

Sequential File

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

100	
110	

110	
120	

120	

Sparse Index

10	
30	
50	
70	

90	
110	
130	
150	

170	
190	
210	
230	

Sequential File

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

Sparse 2nd level

10	-
90	
170	
250	

330	-
410	
490	
570	

10	-
30	
50	
70	

90	-
110	
130	
150	

170	-
190	
210	
230	

Sequential File

10	
20	

30	
40	

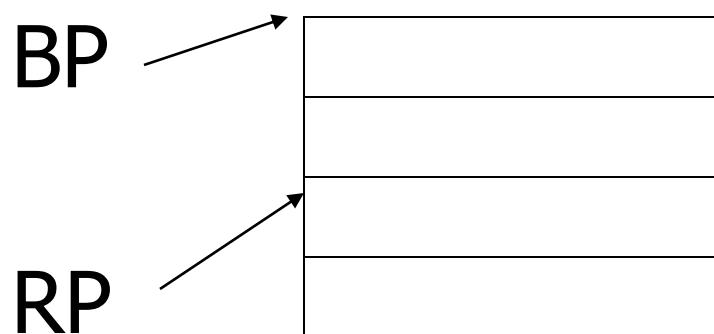
50	
60	

70	
80	

90	
100	

Notes on pointers:

(1) Block pointer (sparse index) can be smaller than record pointer



Sparse vs. Dense Tradeoff

- Sparse: Less index space per record
can keep more of index in memory
- Dense: Can tell if any record exists
without accessing file

(Later:

- sparse better for insertions
- dense needed for secondary indexes)

Terms

- Index sequential file
- Search key (\neq primary key)
- Primary index (on ordering field)
- Secondary index (on non-ordering field)
- Dense index (all Search Key values in)
- Sparse index
- Multi-level index

Next:

- Duplicate keys
- Deletion/Insertion
- Secondary indexes

Duplicate keys

10	
10	

10	
20	

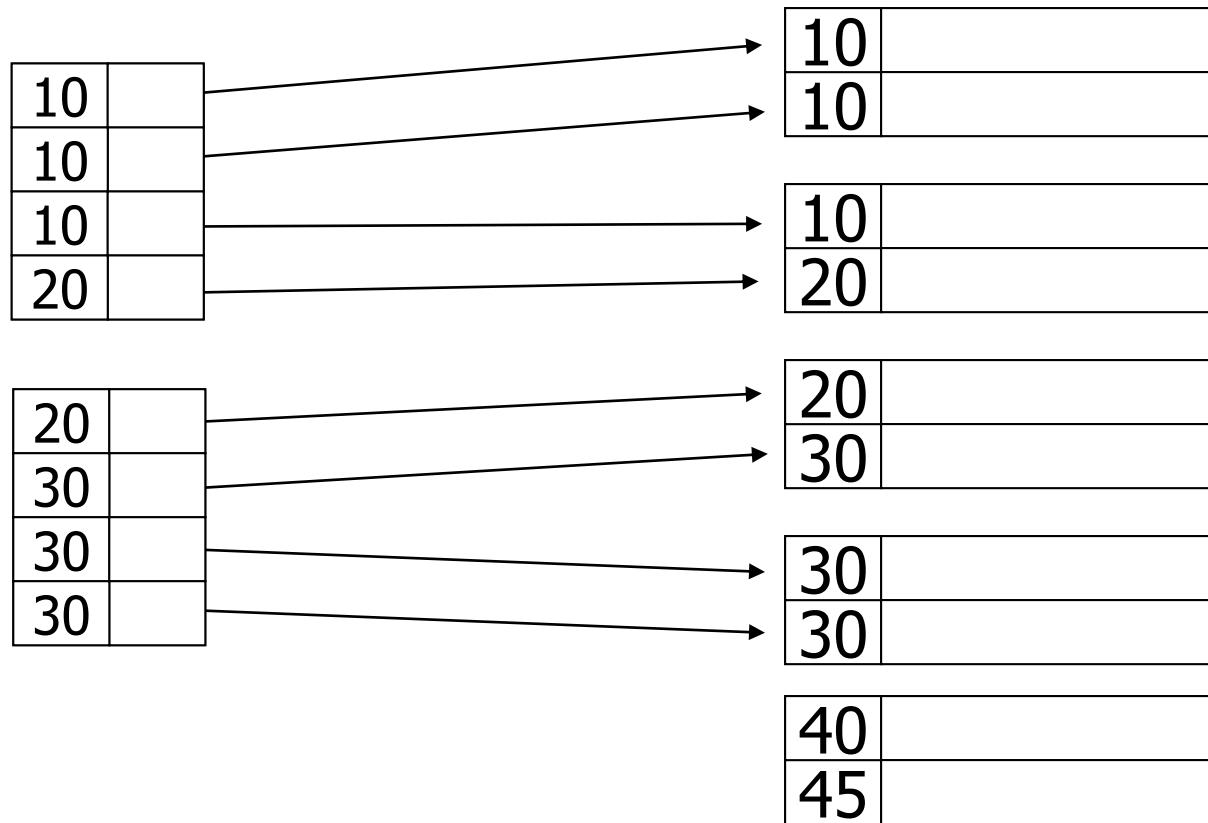
20	
30	

30	
30	

40	
45	

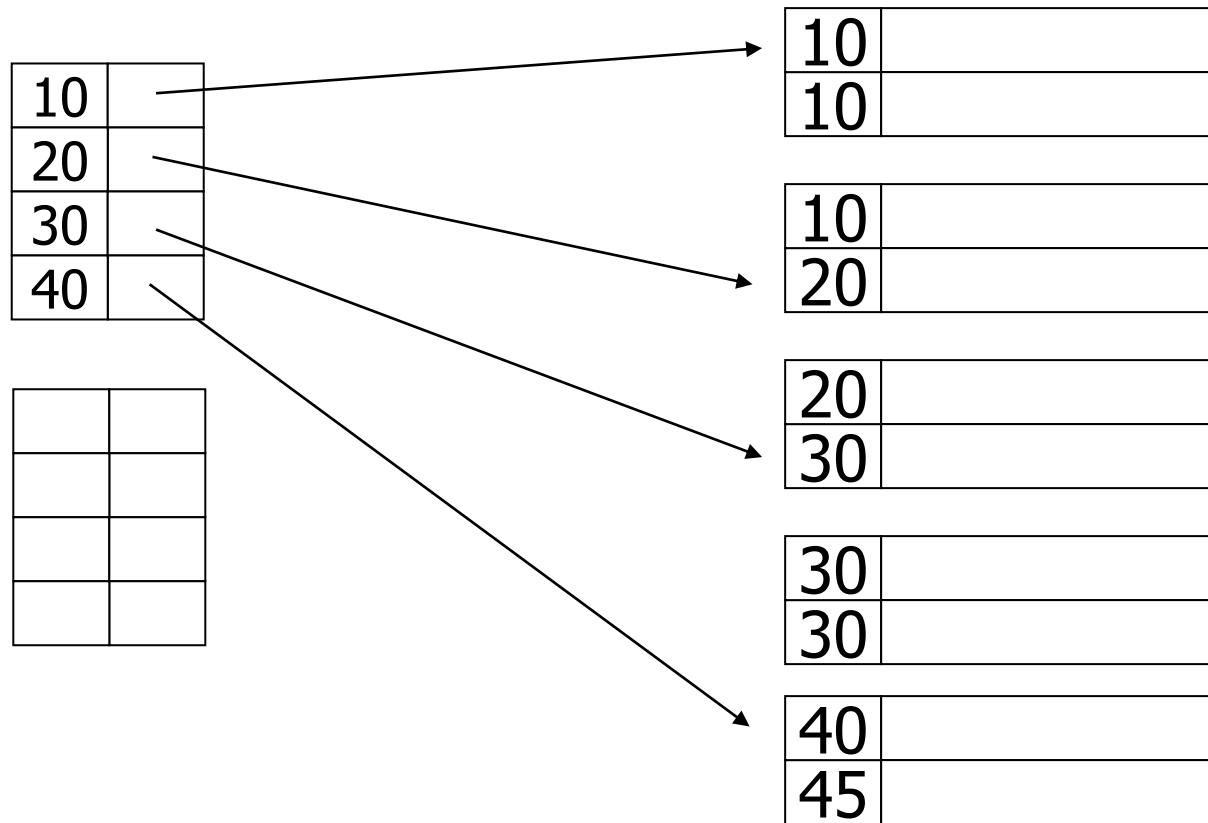
Duplicate keys

Dense index, one way to implement?



Duplicate keys

Dense index, better way?



Duplicate keys

Sparse index, one way?

careful if looking
for 20 or 30!

10	
10	
20	
30	

10	
10	

10	
20	

20	
30	

30	
30	

40	
45	

Duplicate keys

Sparse index, another way?

- place **first new key from block**

should
this be
40?

10	
20	
30	
30	

10	
10	

10	
20	

20	
30	

30	
30	

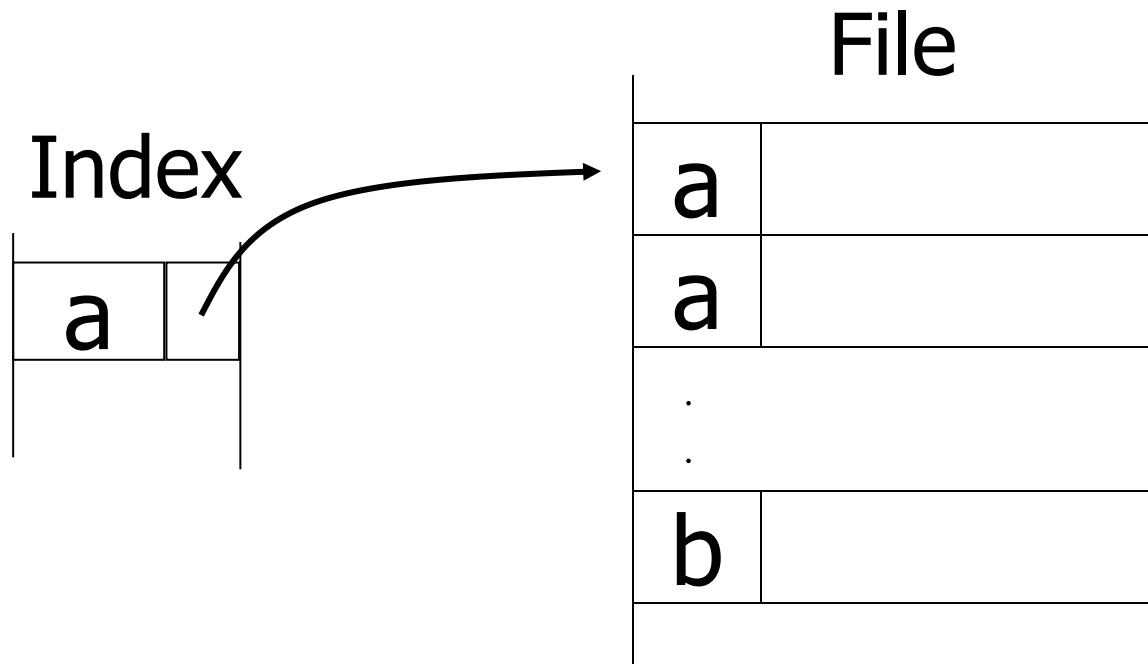
40	
45	

✗

Summary

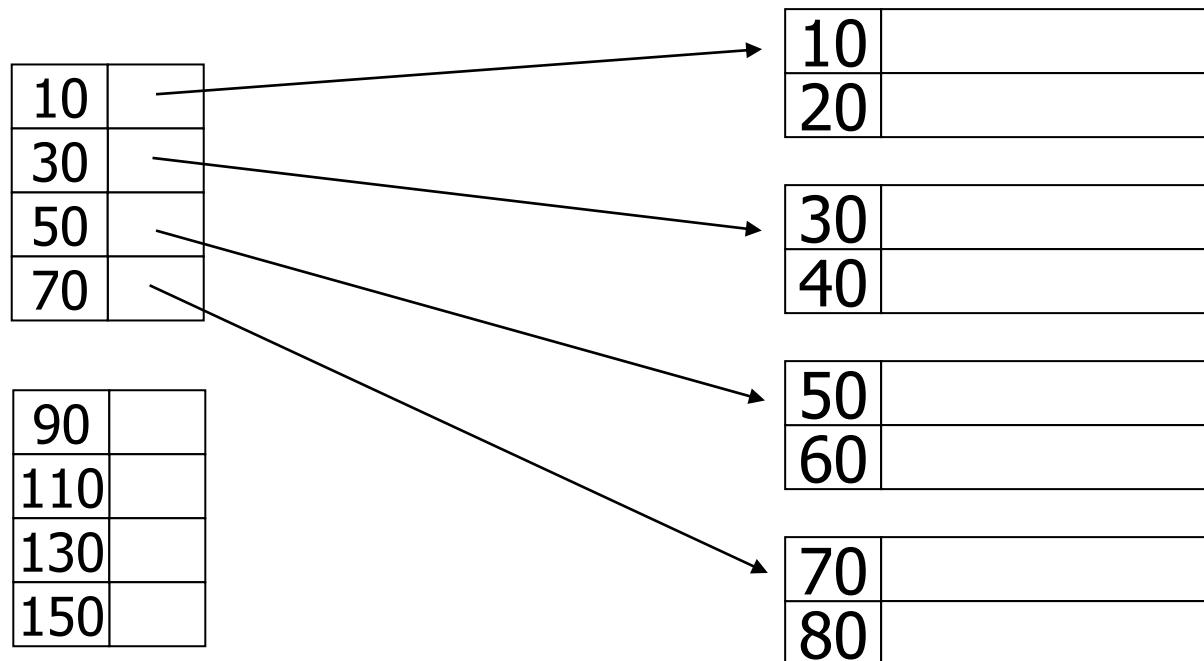
Duplicate values, primary index

- Index may point to first instance of each value only



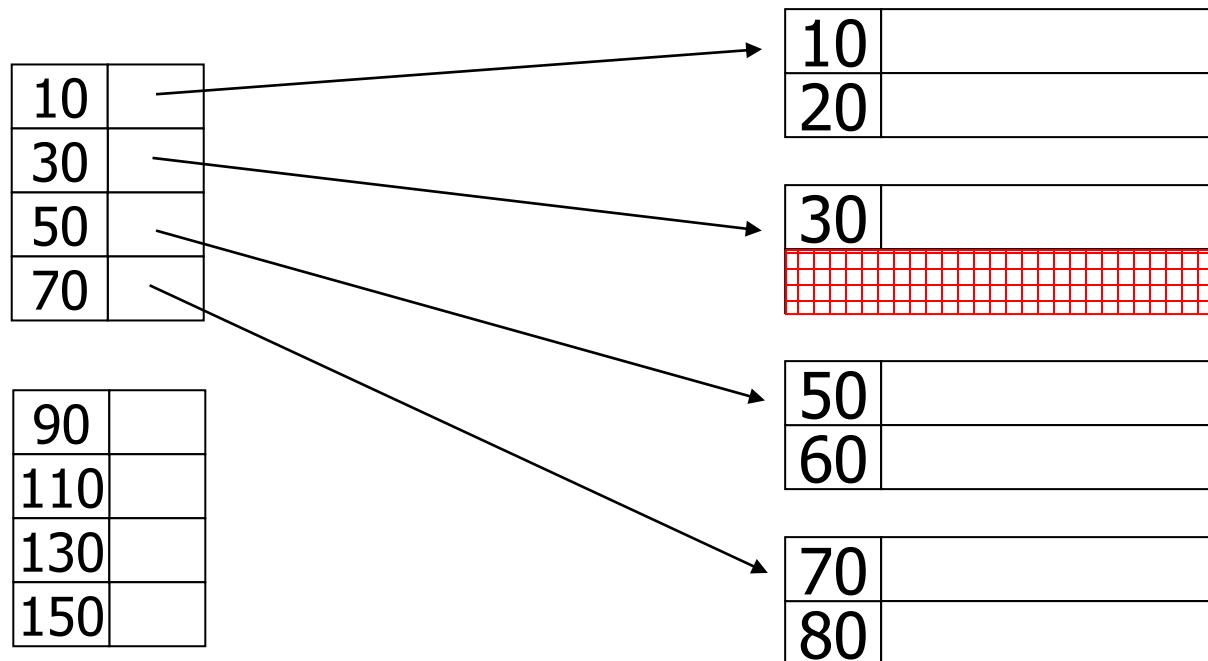
Deletion from sparse index

– delete record 40



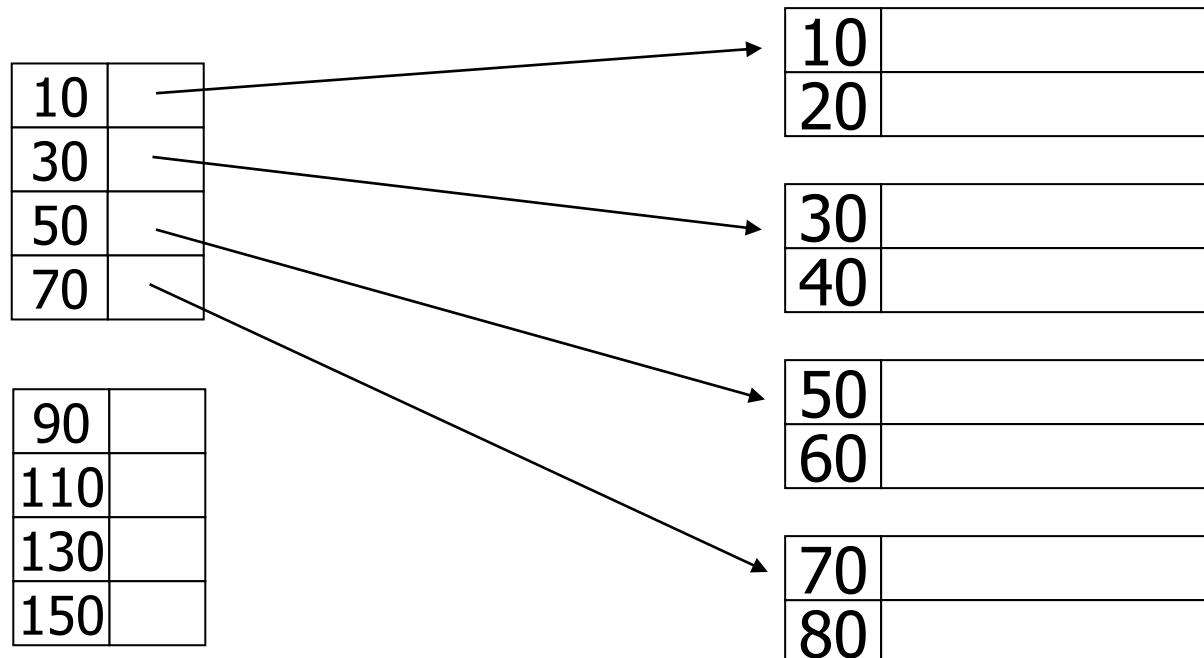
Deletion from sparse index

– delete record 40



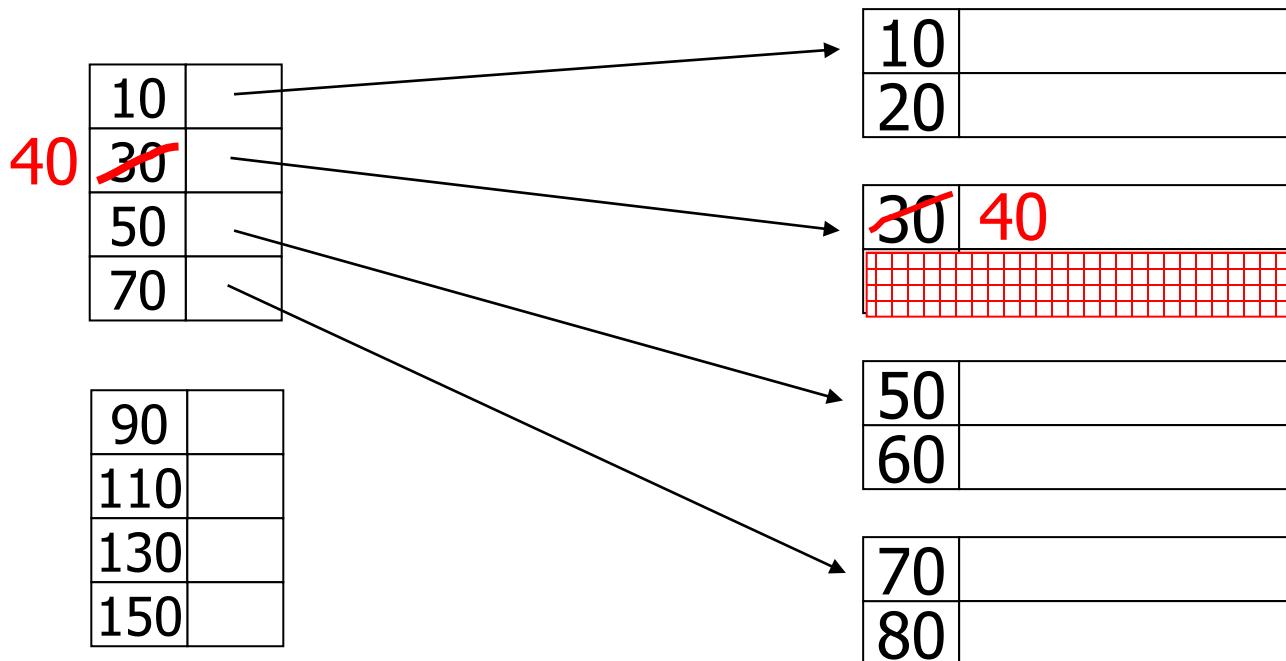
Deletion from sparse index

– delete record 30



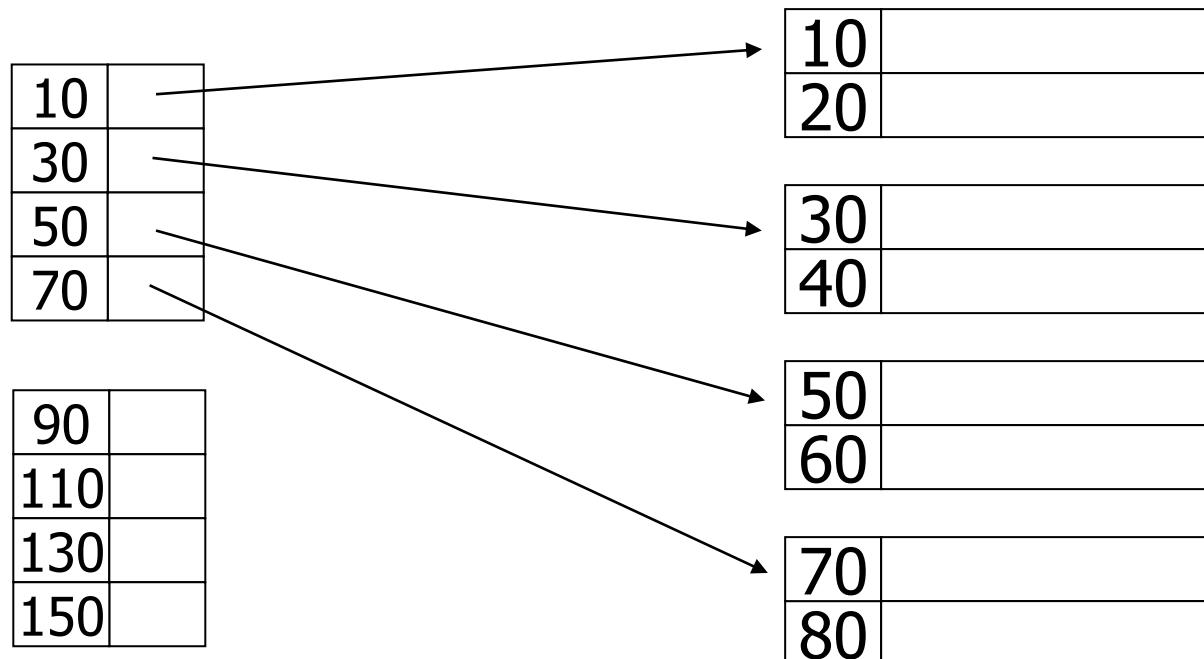
Deletion from sparse index

– delete record 30



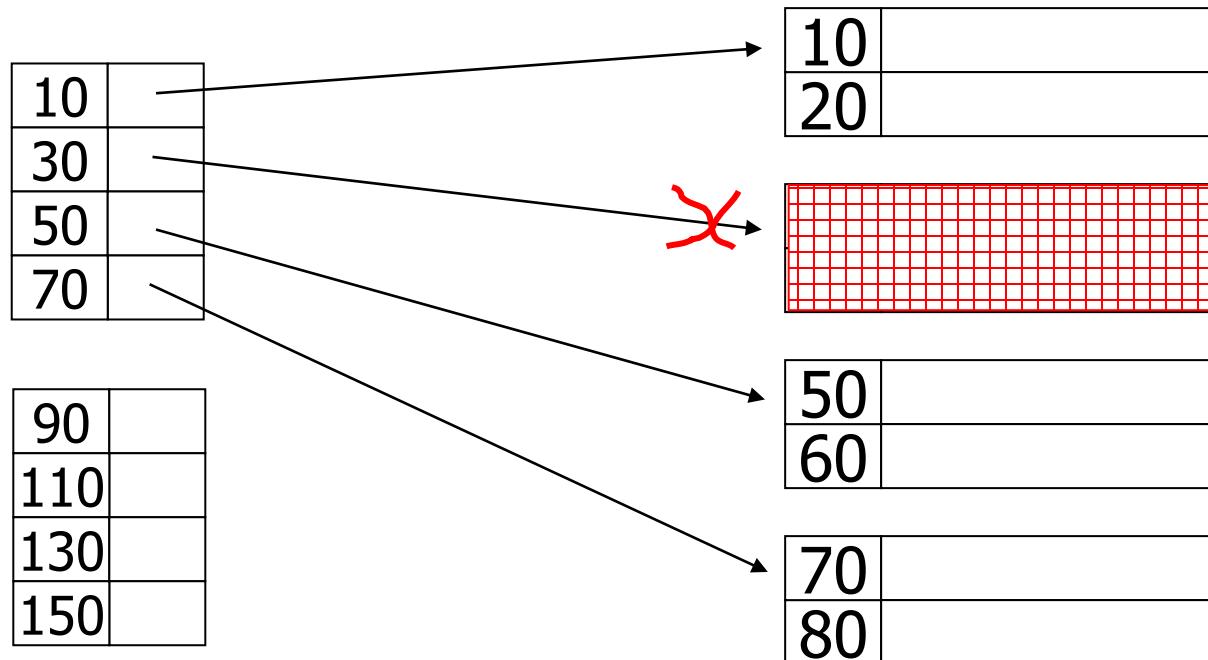
Deletion from sparse index

– delete records 30 & 40



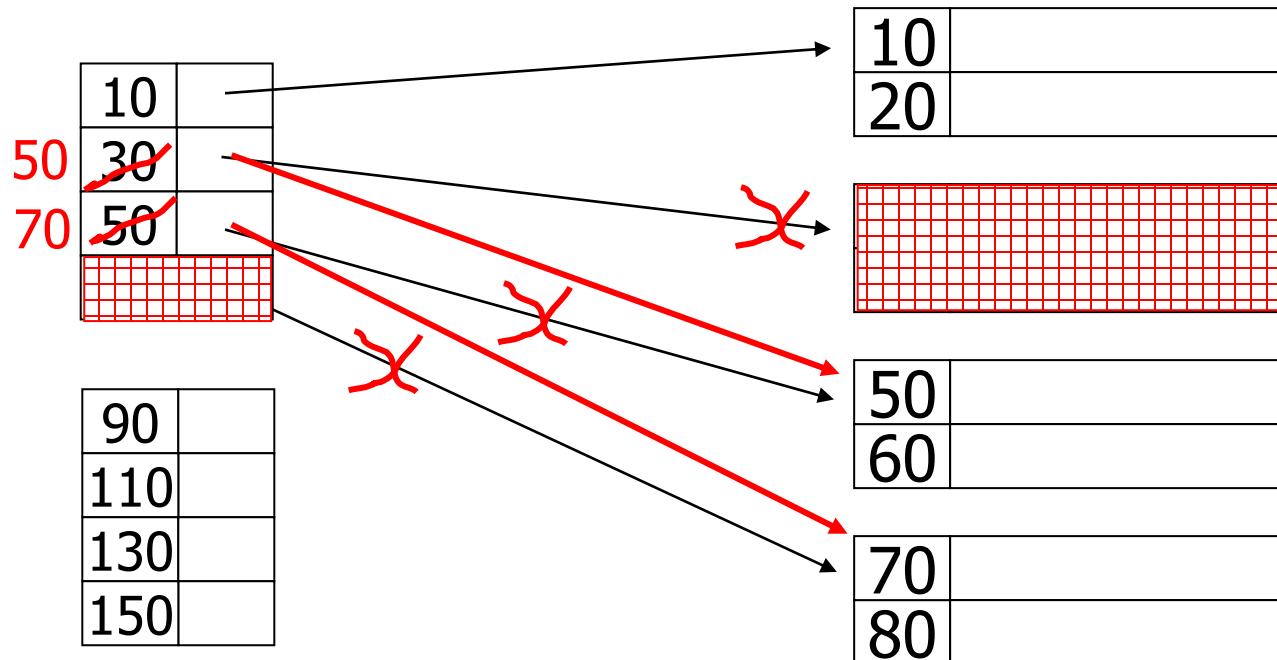
Deletion from sparse index

– delete records 30 & 40



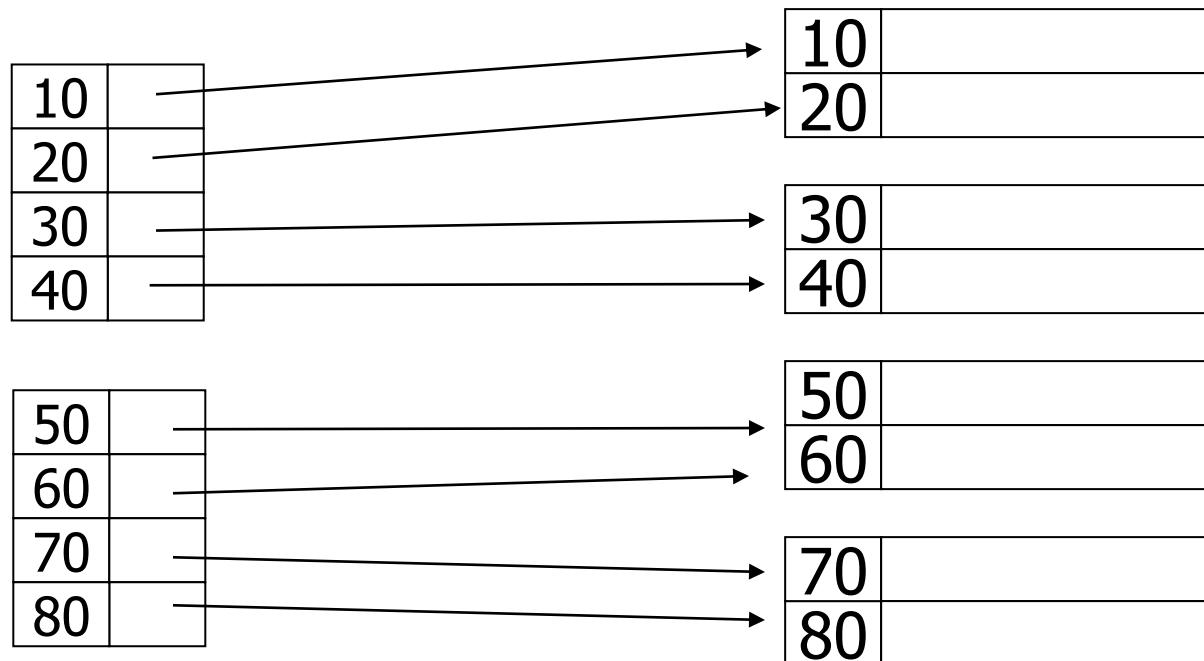
Deletion from sparse index

– delete records 30 & 40



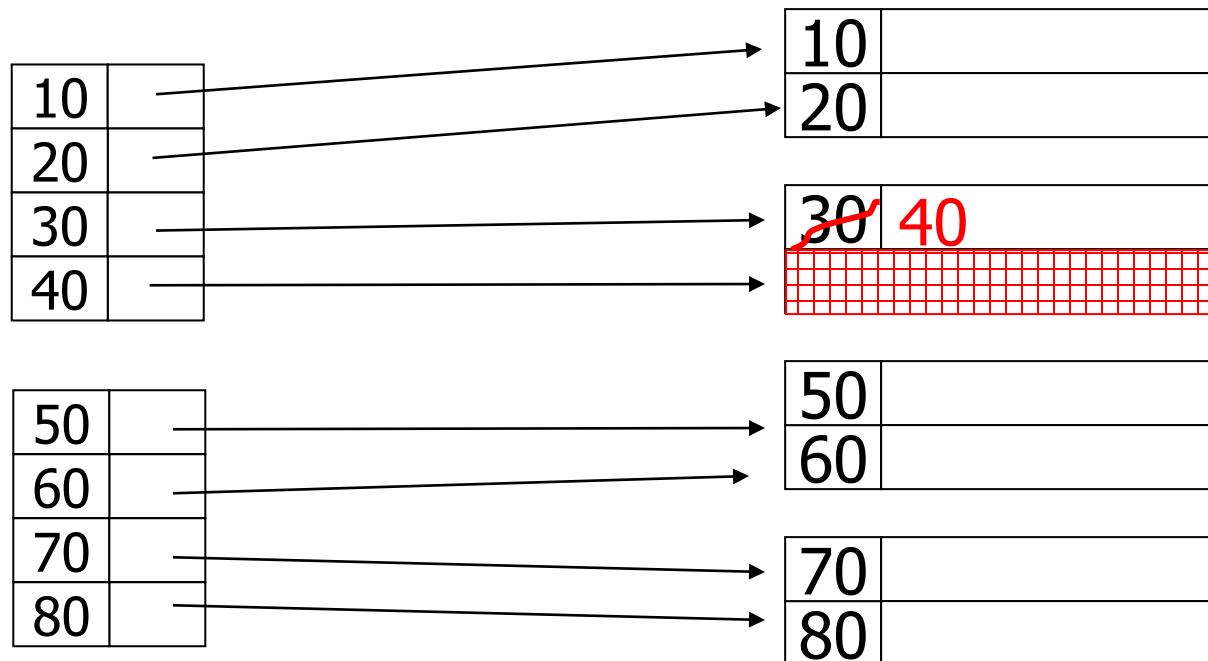
Deletion from dense index

– delete record 30



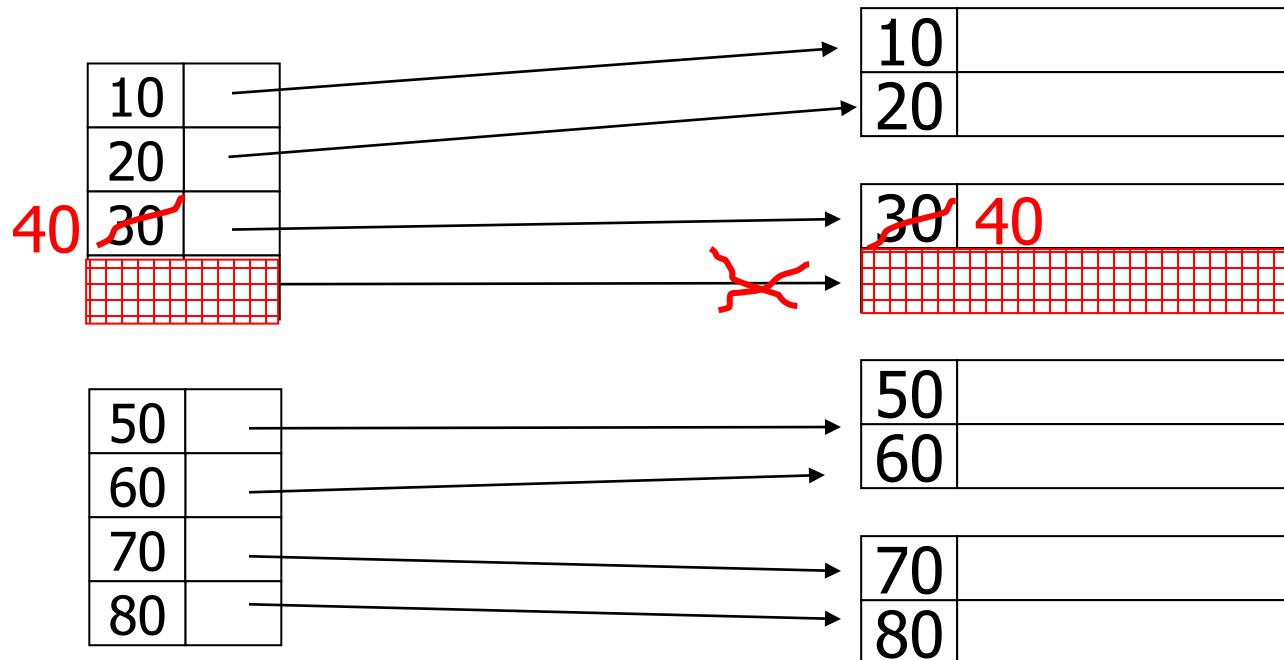
Deletion from dense index

– delete record 30



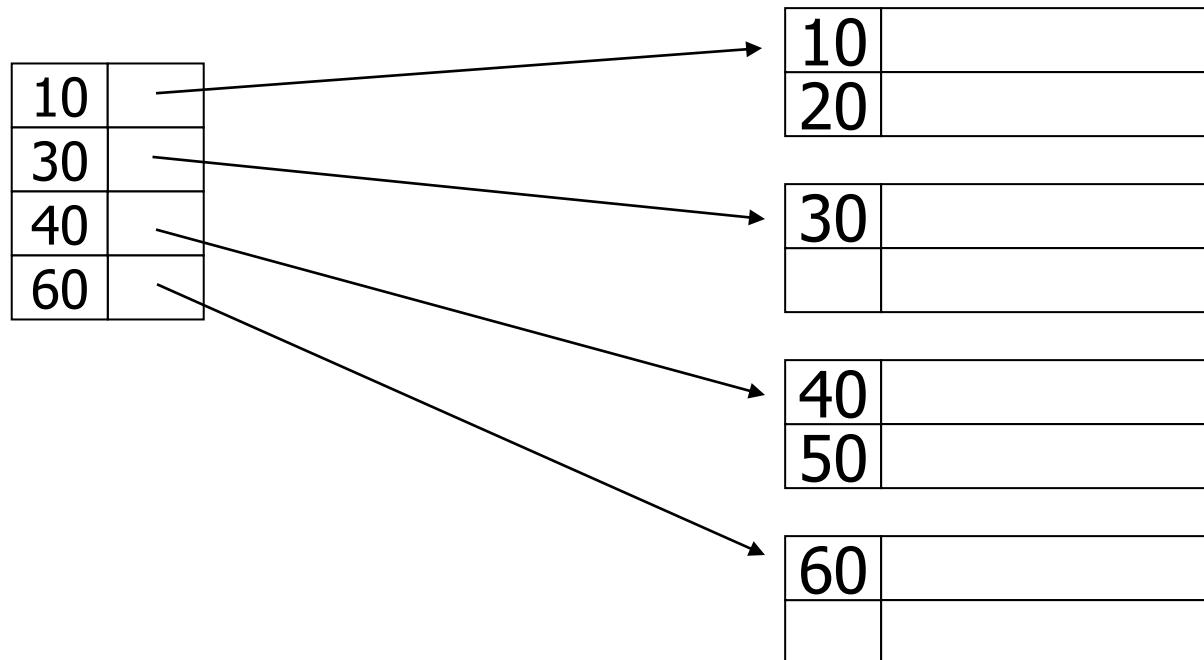
Deletion from dense index

– delete record 30



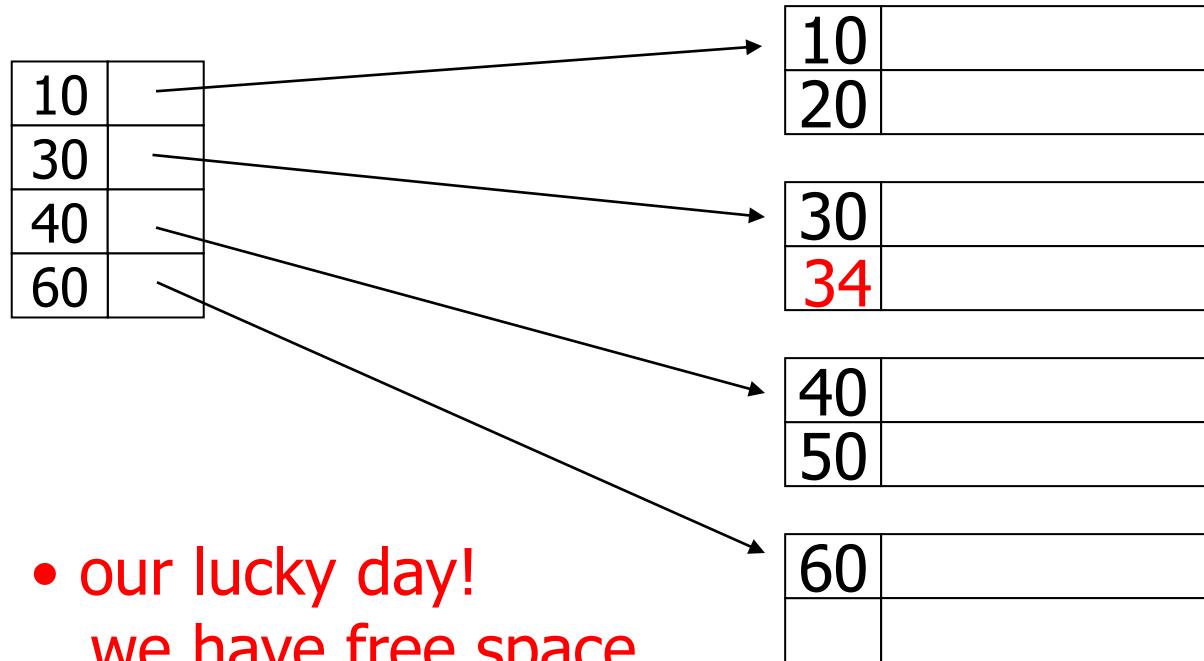
Insertion, sparse index case

– insert record 34



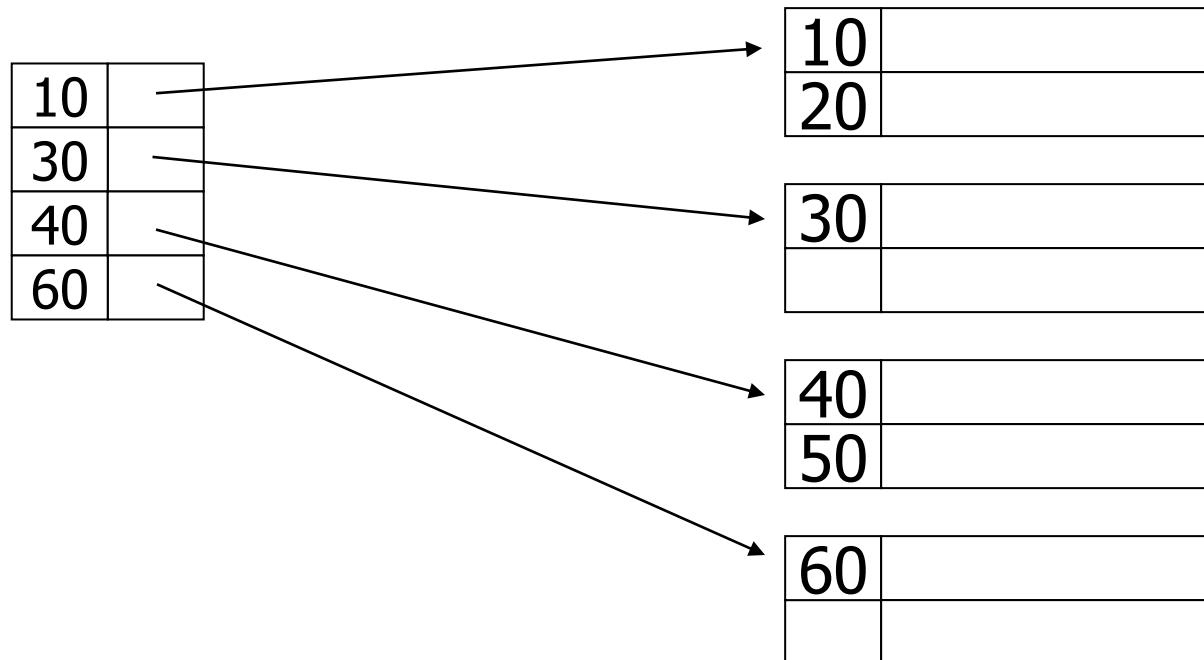
Insertion, sparse index case

– insert record 34



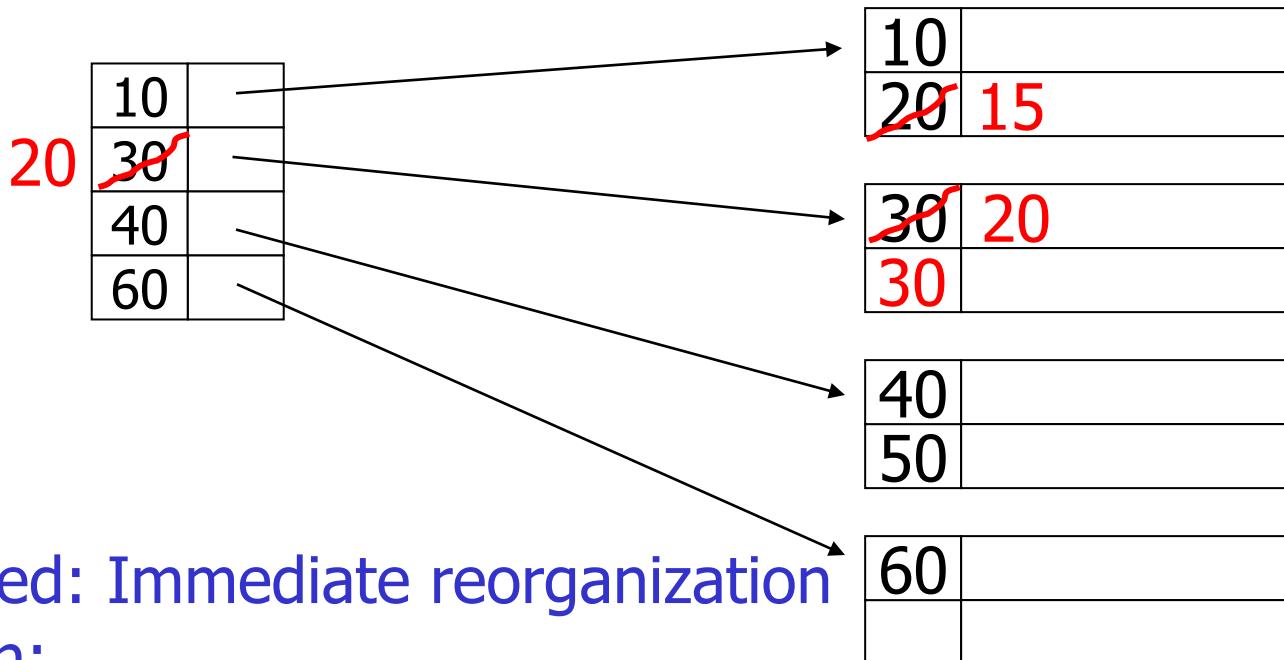
Insertion, sparse index case

– insert record 15



Insertion, sparse index case

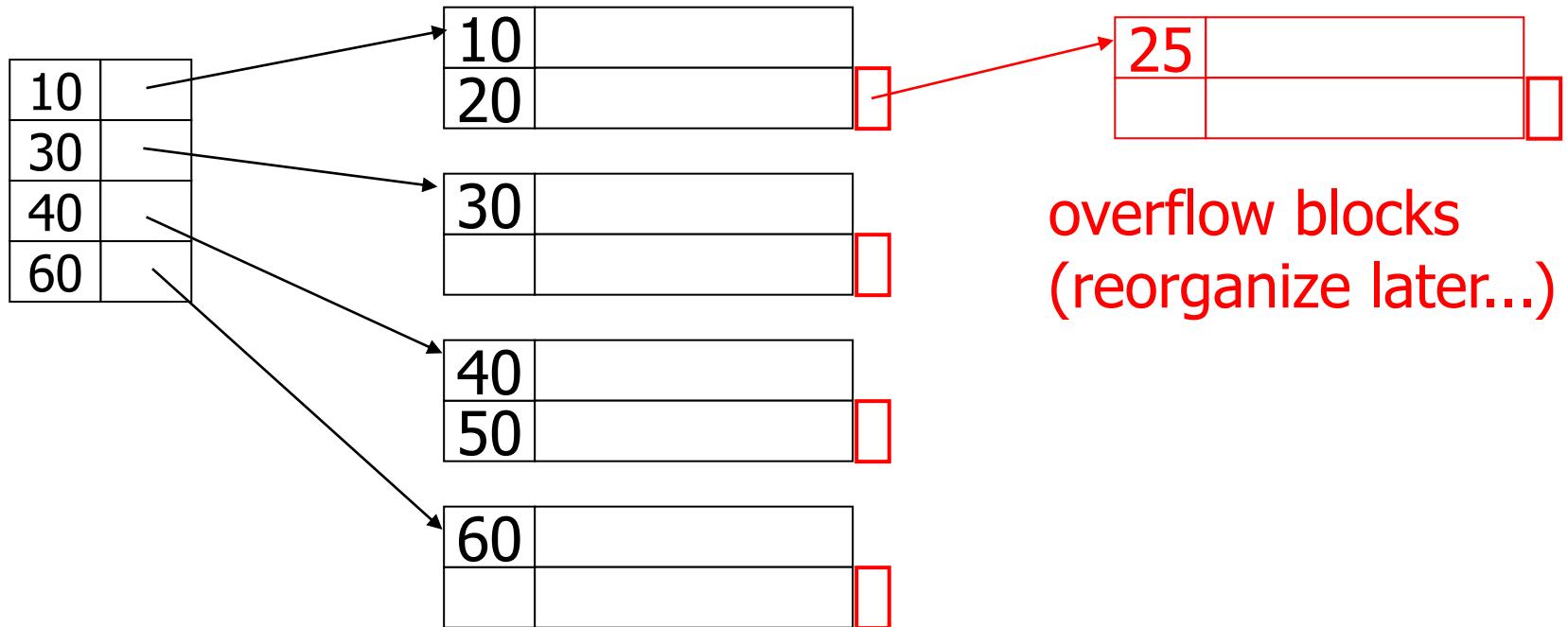
– insert record 15



- Illustrated: Immediate reorganization
- Variation:
 - insert new block (chained file)
 - update index

Insertion, sparse index case

– insert record 25

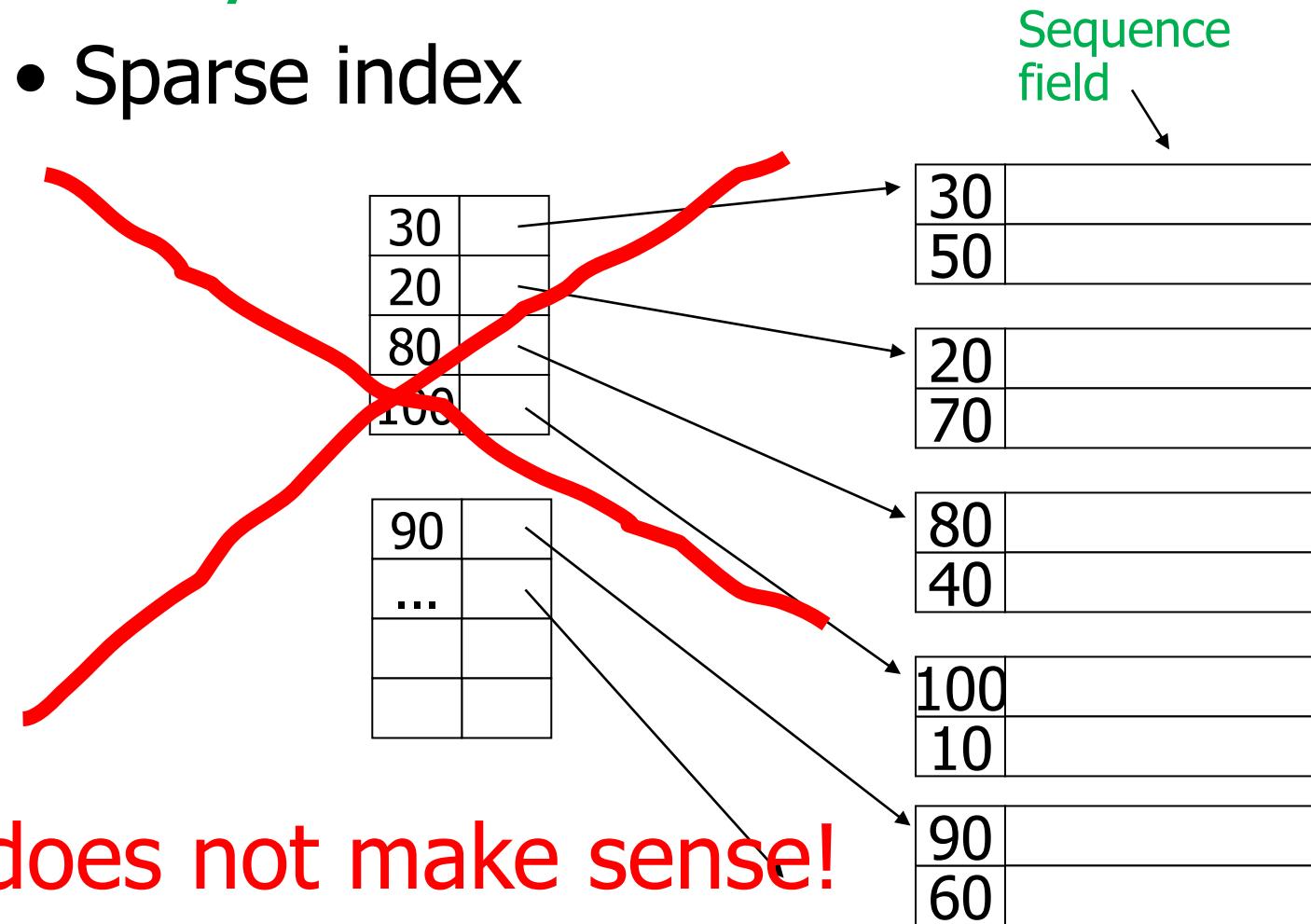


Insertion, dense index case

- Similar
- Often more expensive . . .

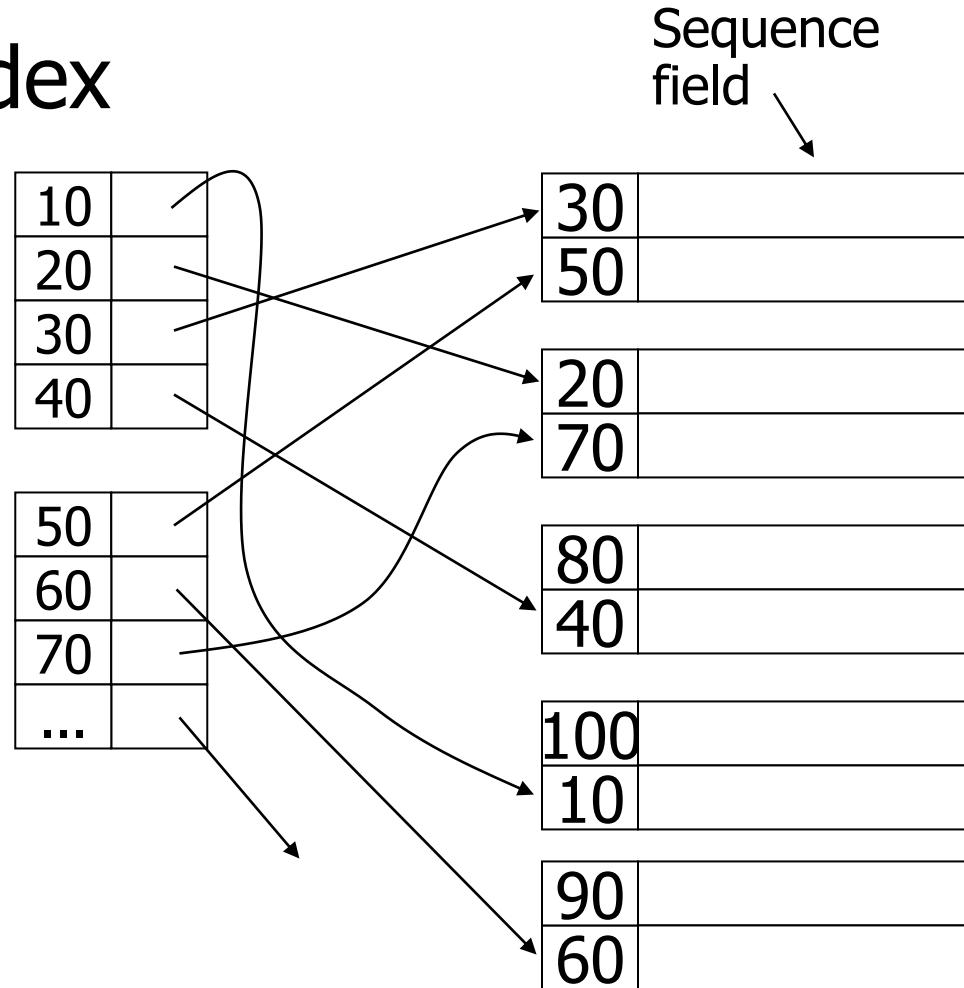
Secondary indexes

- Sparse index



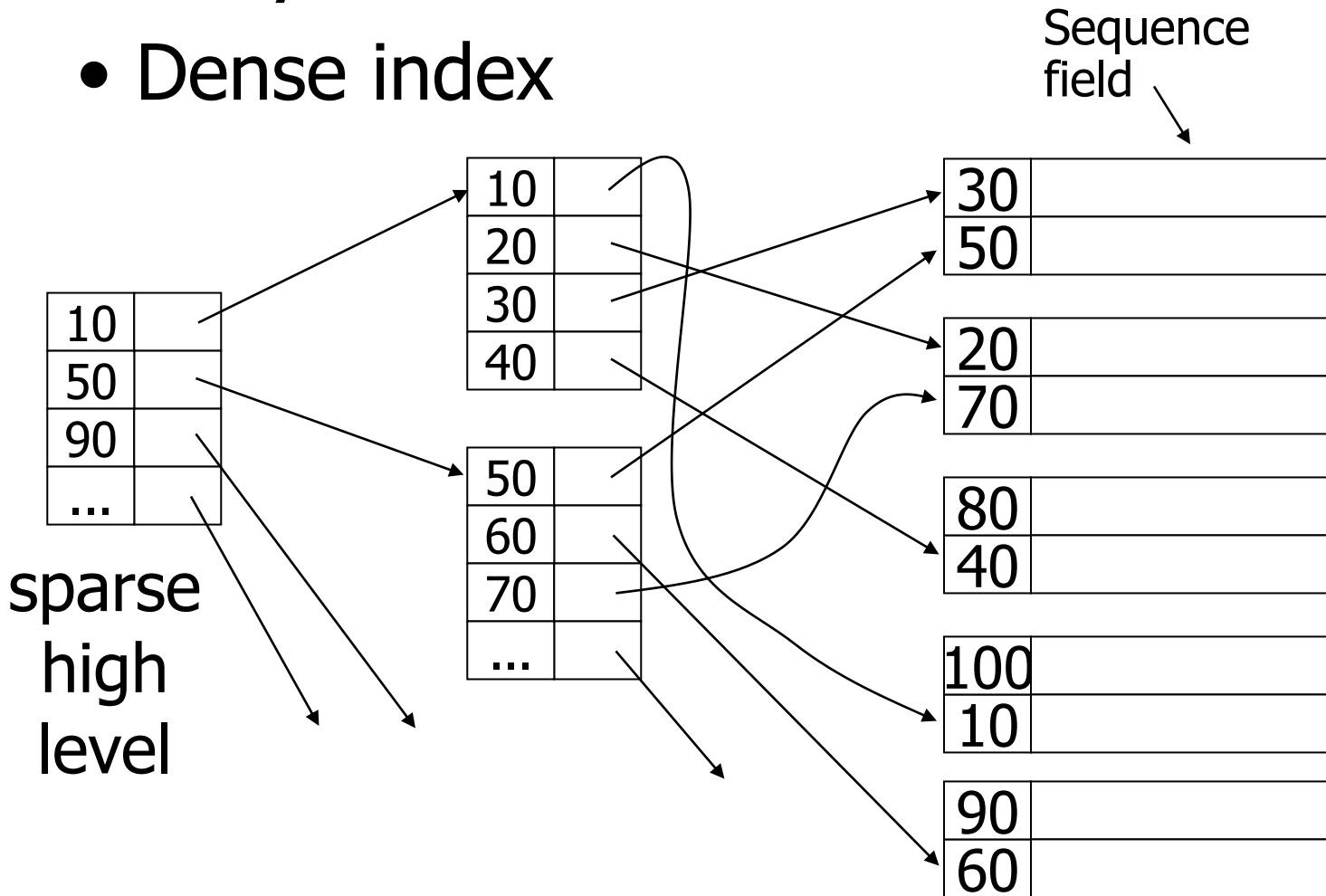
Secondary indexes

- Dense index



Secondary indexes

- Dense index



With secondary indexes:

- Lowest level is dense
- Other levels are sparse

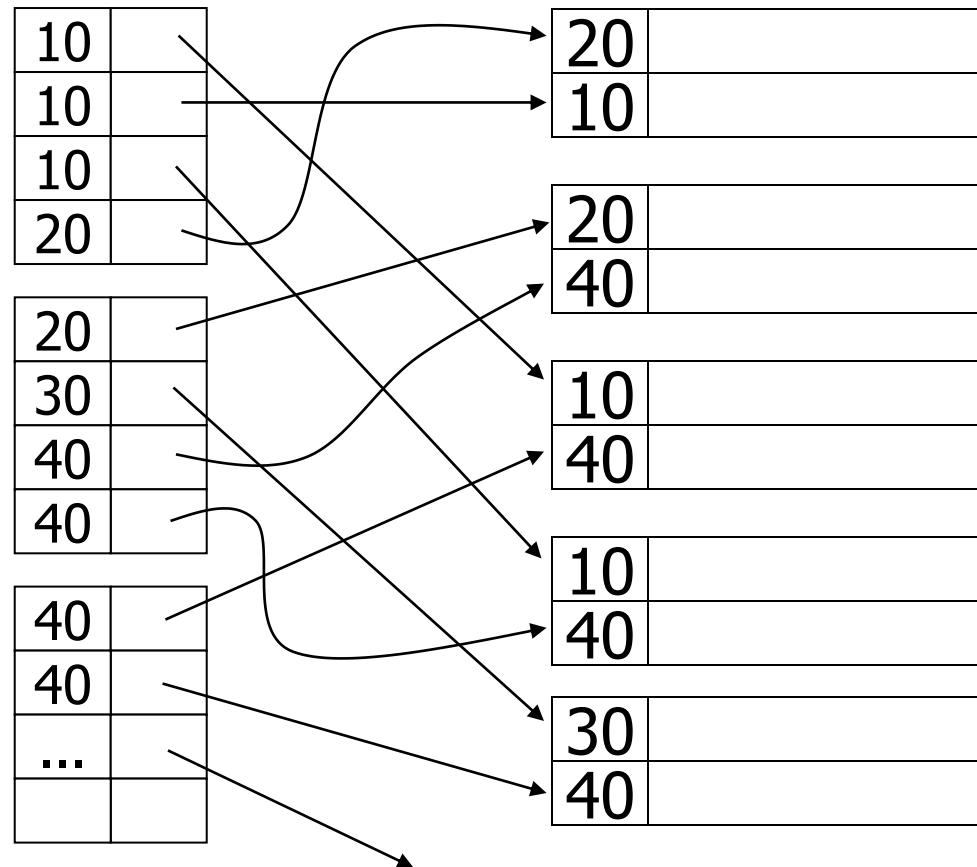
Also: Pointers are record pointers
(not block pointers)

Duplicate values & secondary indexes

one option...

Problem:
excess overhead!

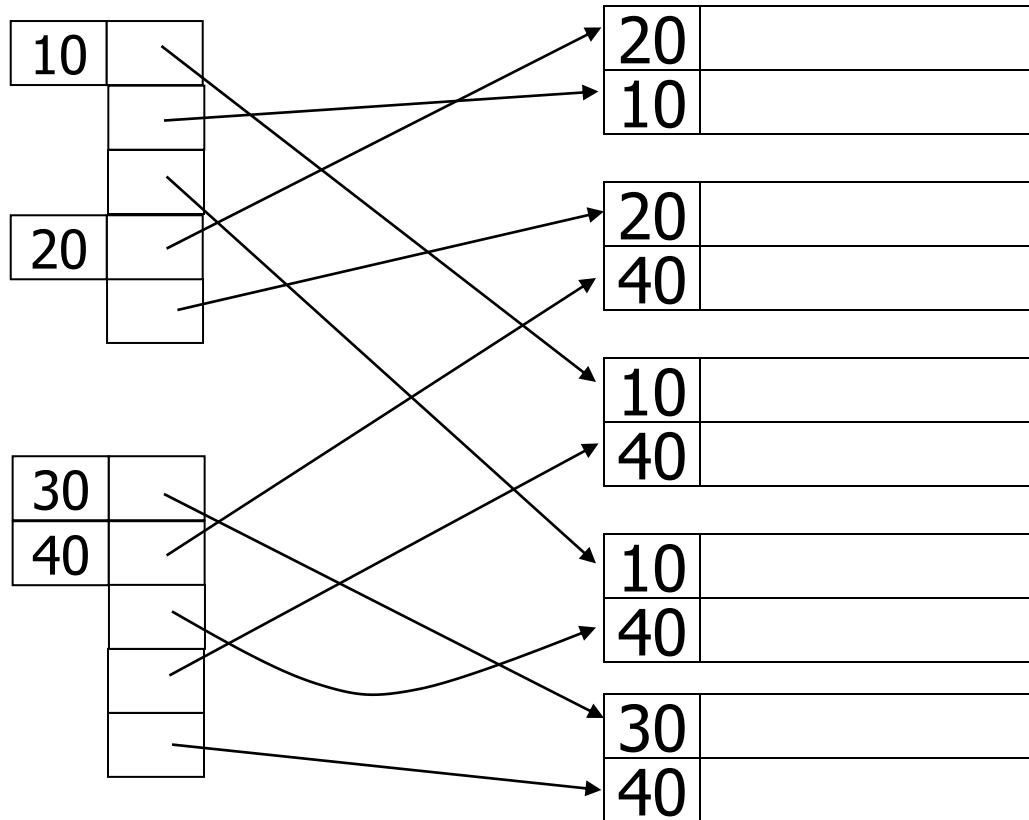
- disk space
- search time



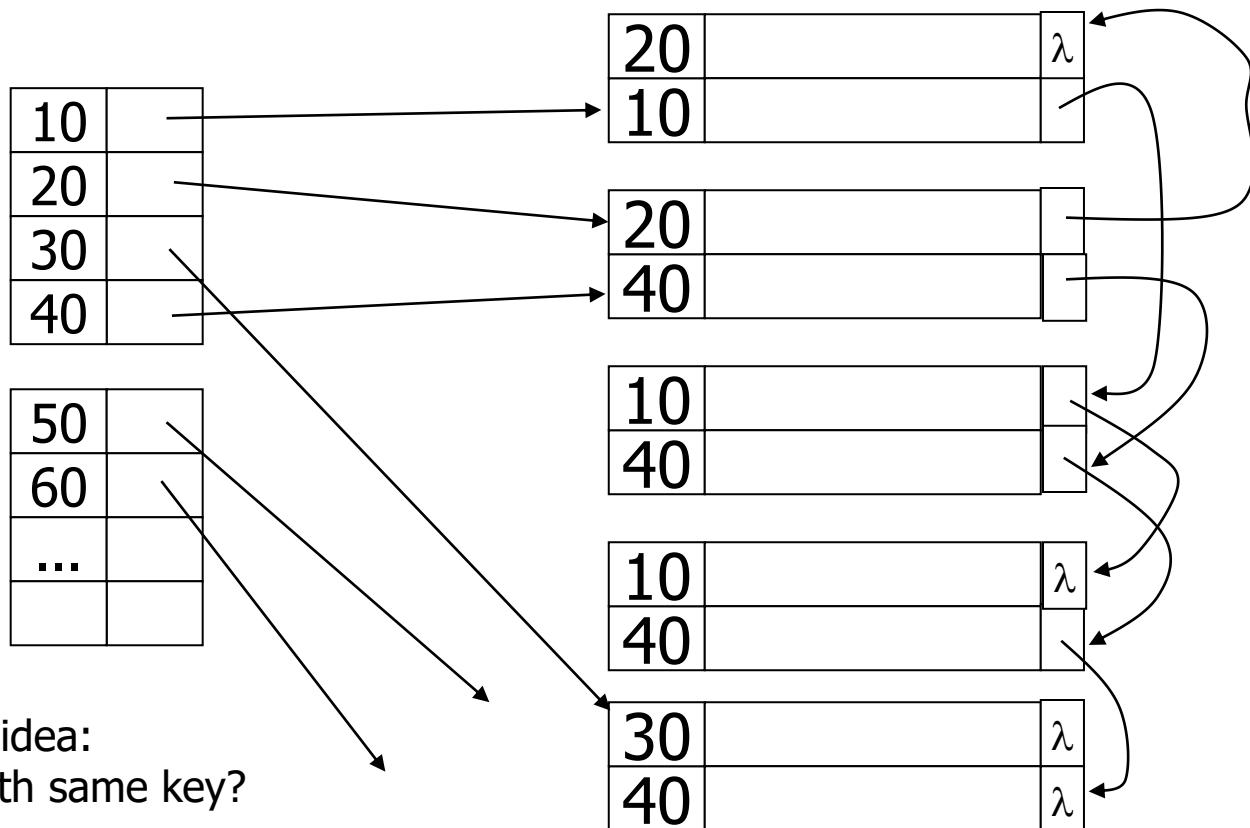
Duplicate values & secondary indexes

another option...

Problem:
variable size
records in
index!



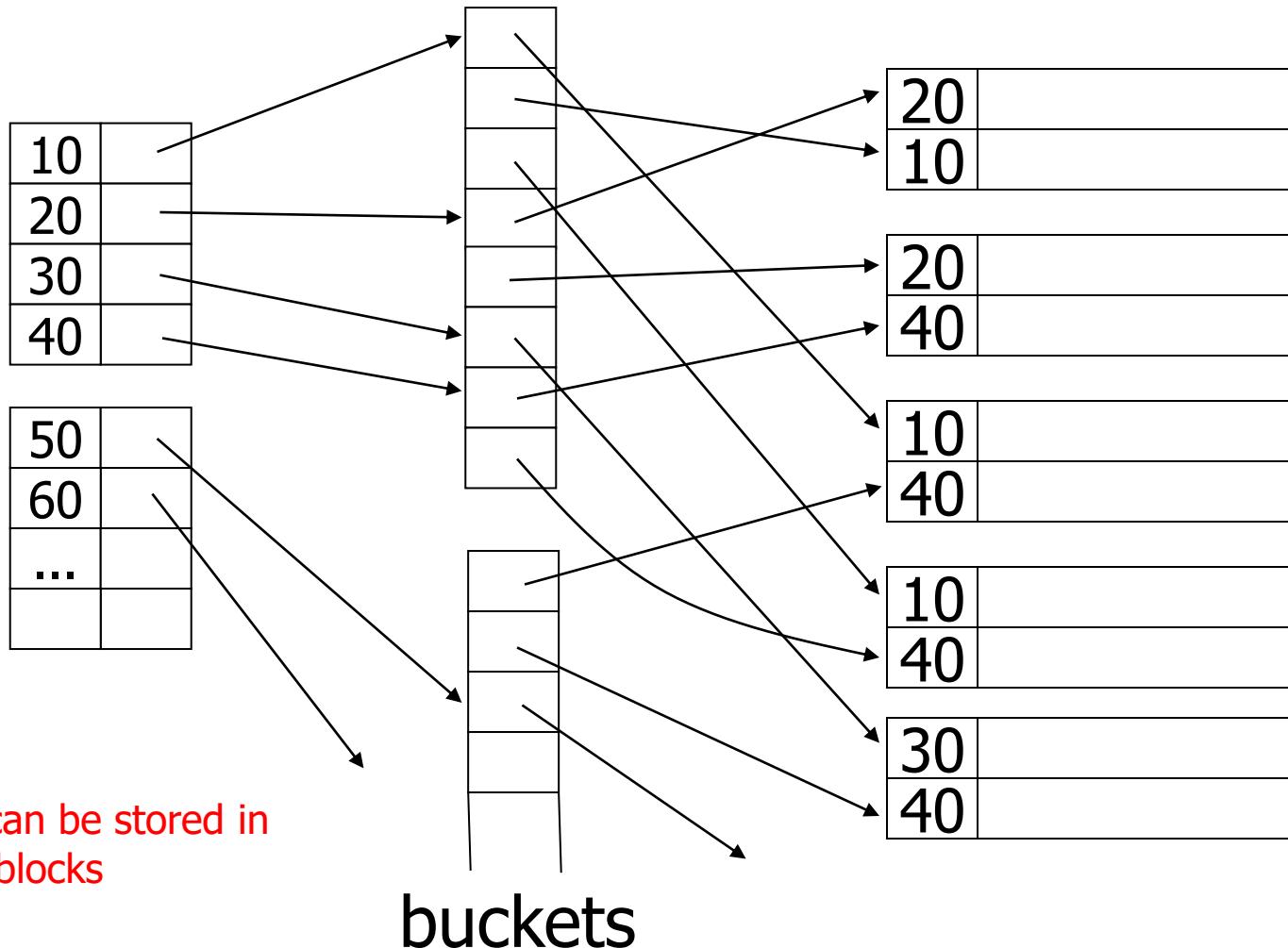
Duplicate values & secondary indexes



Problems:

- Need to add fields to records
- Need to follow chain to know records

Duplicate values & secondary indexes



Pointers can be stored in
separate blocks

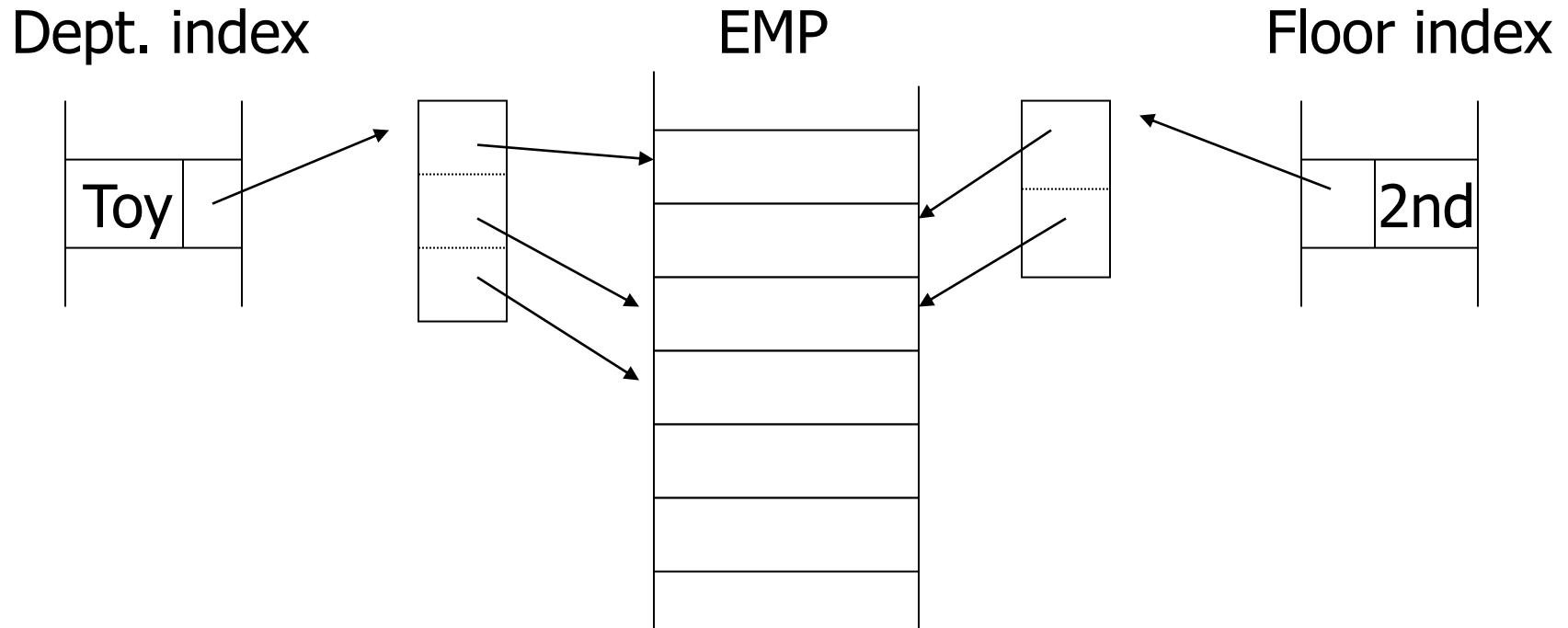
buckets

Why “bucket” idea is useful

<u>Indexes</u>	<u>Records</u>
Name: primary	EMP (name,dept,floor,...)
Dept: secondary	
Floor: secondary	

See the following query

Query: Get employees in
(Toy Dept) \wedge (2nd floor)



→ Intersect toy bucket and 2nd Floor
bucket to get set of matching EMP's

Summary so far

- Conventional index
 - Basic Ideas: sparse, dense, multi-level...
 - Duplicate Keys
 - Deletion/Insertion
 - Secondary indexes

Conventional indexes

Advantage:

- Simple
- Index is sequential file
good for scans

Disadvantage:

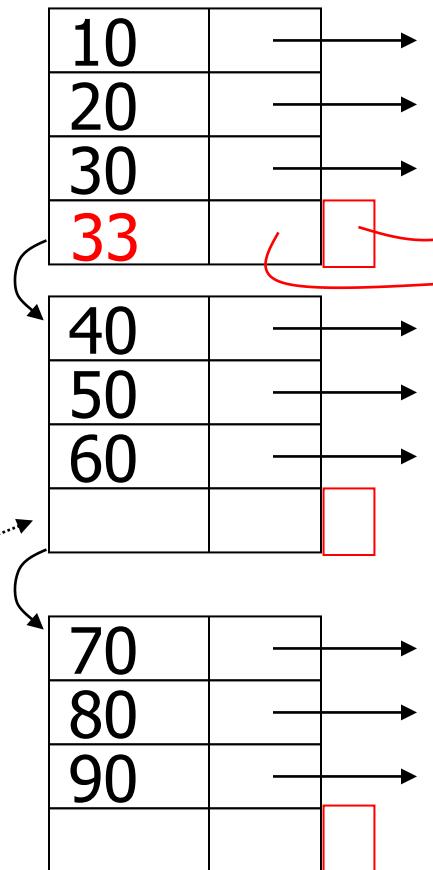
- Inserts expensive, and/or
- Lose sequentiality & balance

Example

Index (sequential)

continuous

free space

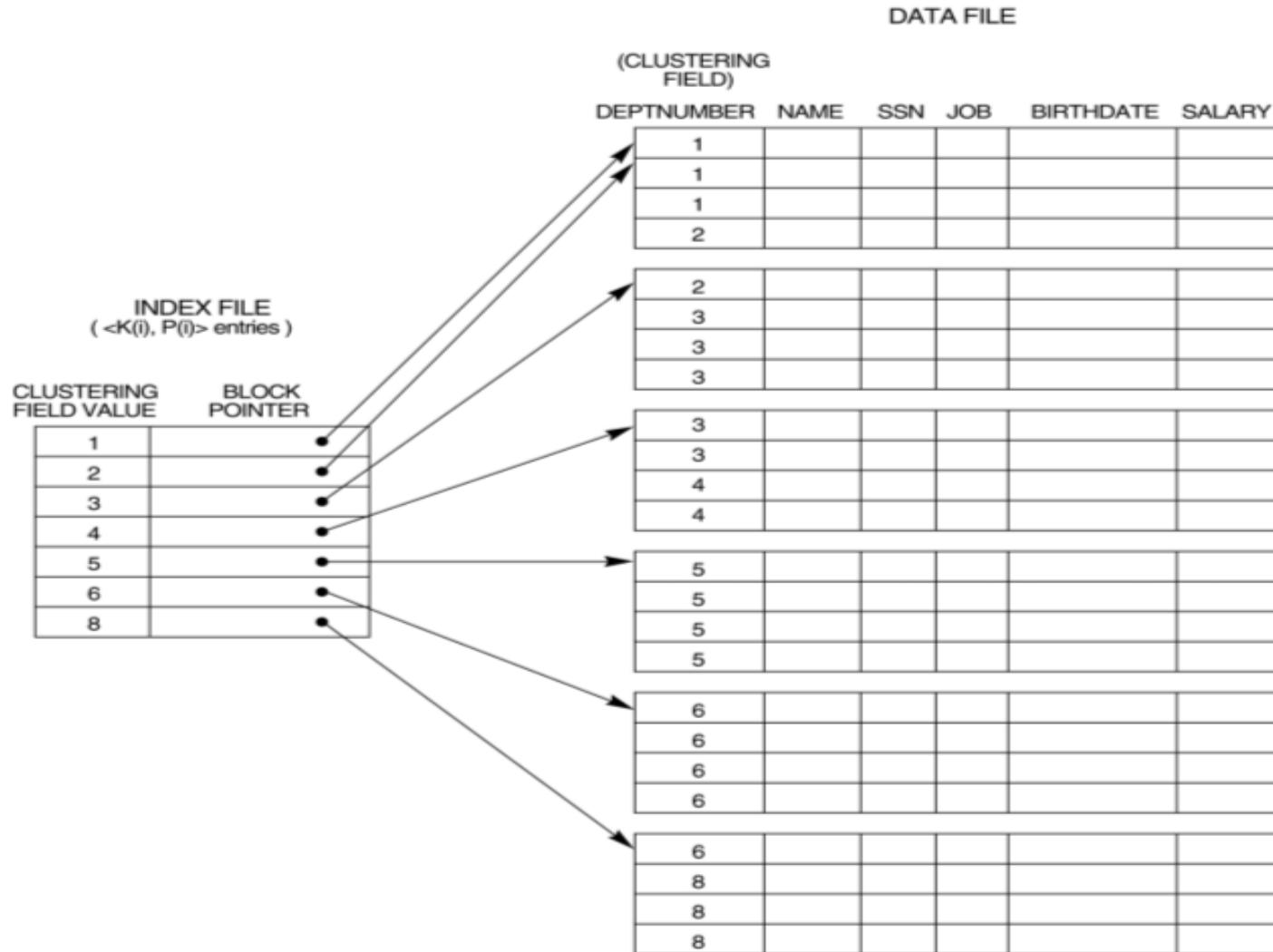


overflow area
(not sequential)

- **Clustering Index**

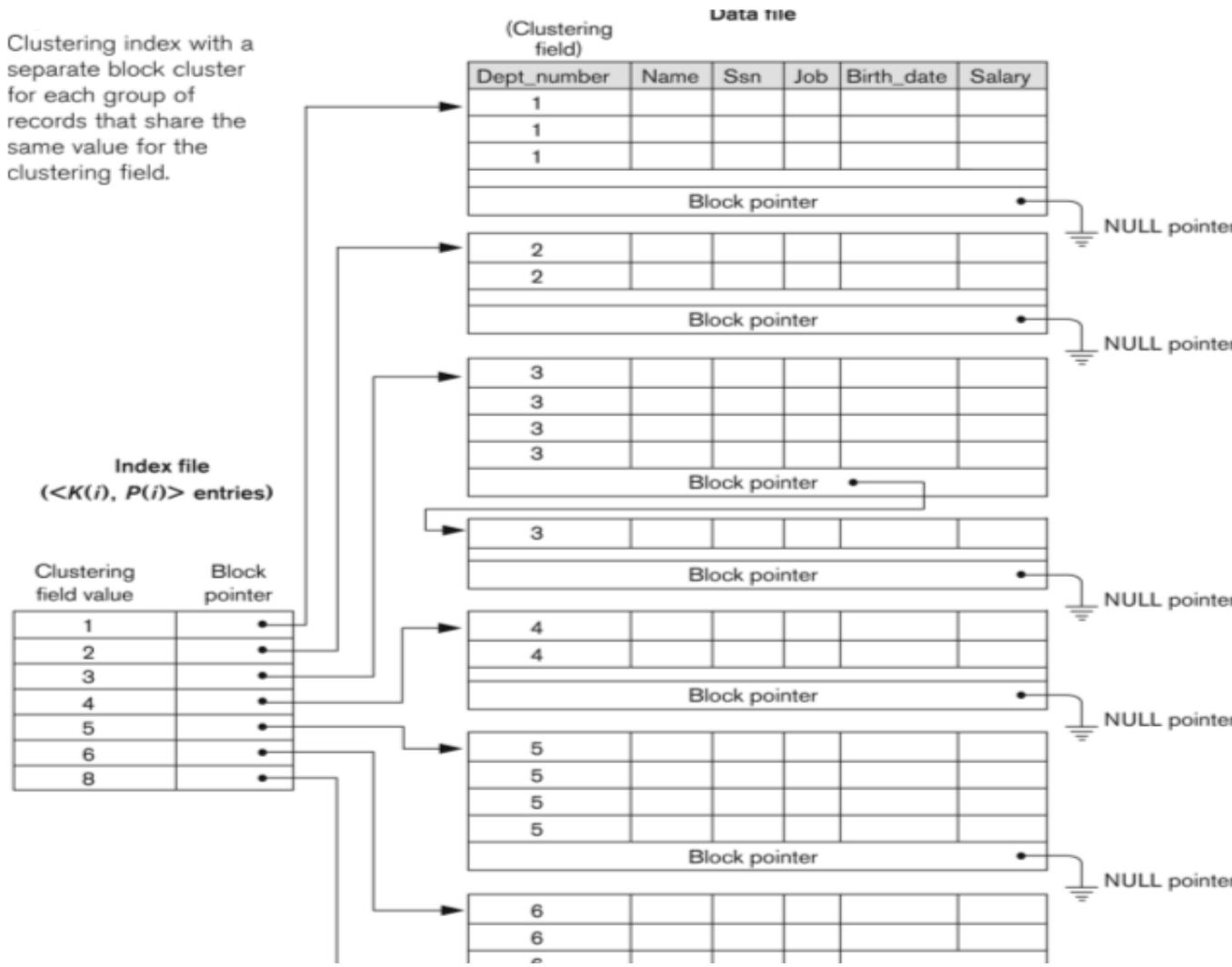
- Defined on an ordered data file
- The **data** file is **ordered on a *non-key field***.
- Includes **one index entry *for each distinct value*** of the field; the index entry points to the first data block that contains records with that field value.
- It is an example of ***non-dense index*** where Insertion and Deletion is relatively straightforward with a clustering index.

- Clustering Index



- Clustering Index version 2

Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



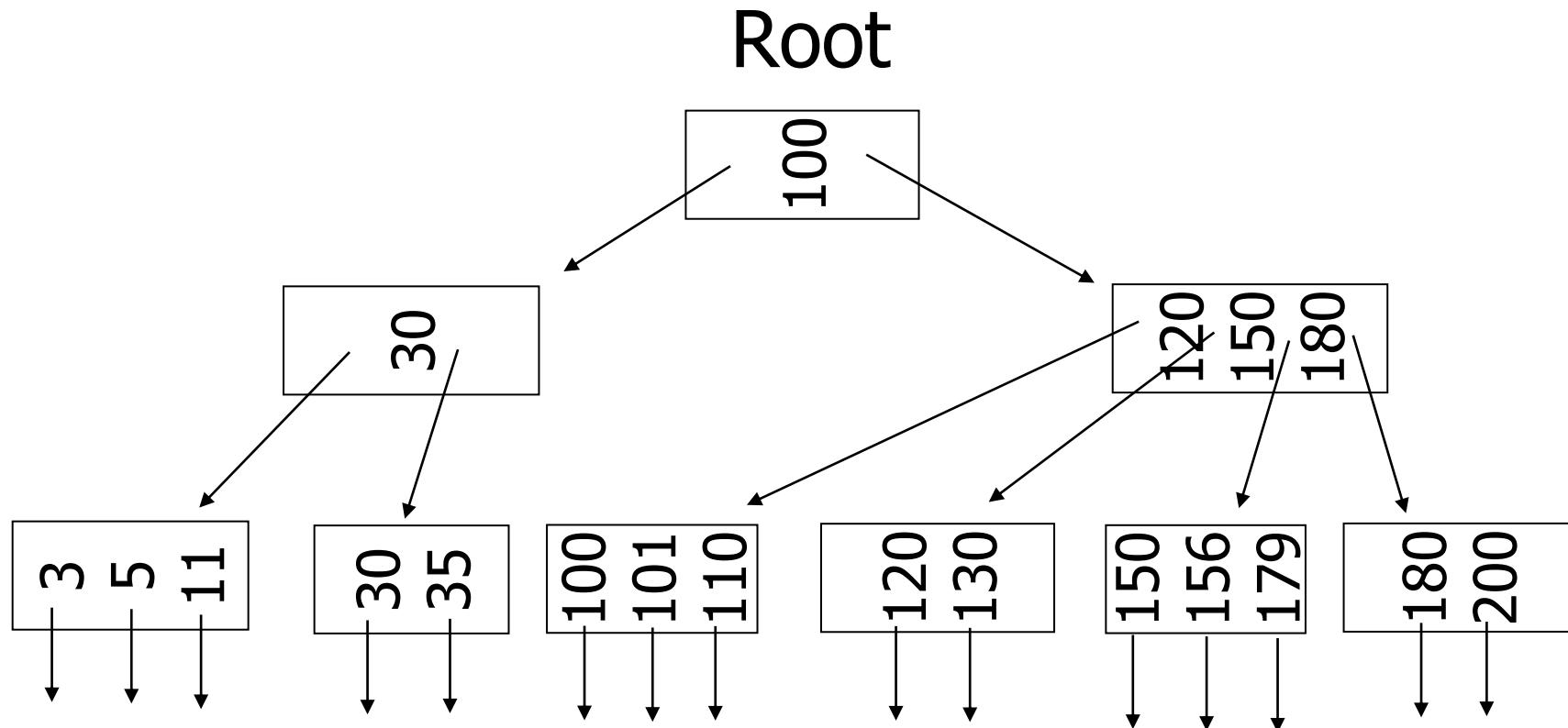
Outline:

- Conventional indexes
- B-Trees \Rightarrow NEXT
- Hashing schemes

- NEXT: Another type of index
 - Give up on sequentiality of index
 - Try to get “balance”

B+Tree Example

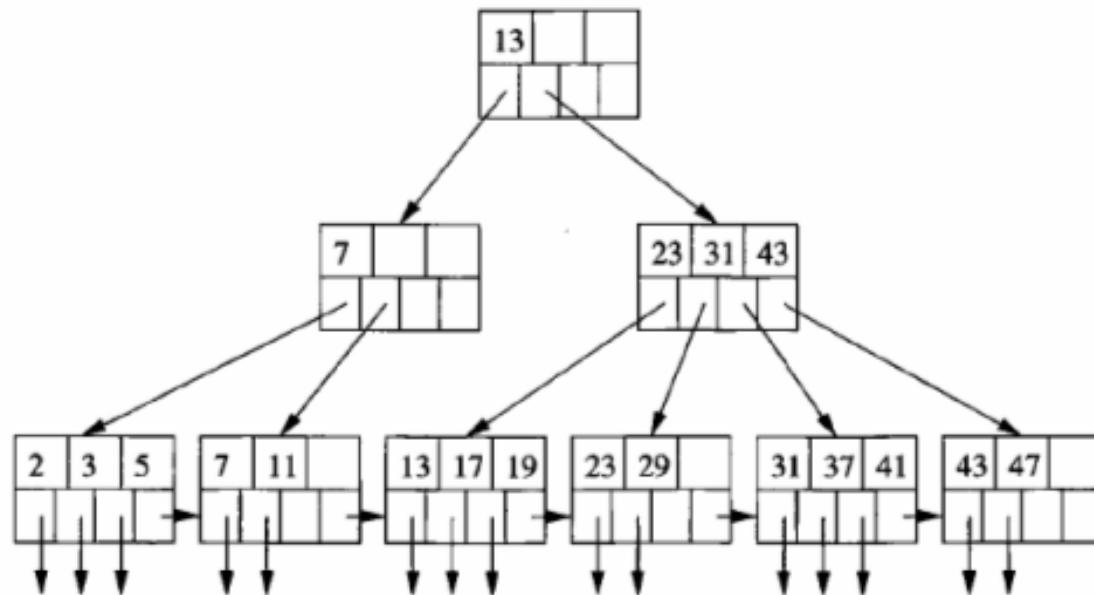
$n=3$



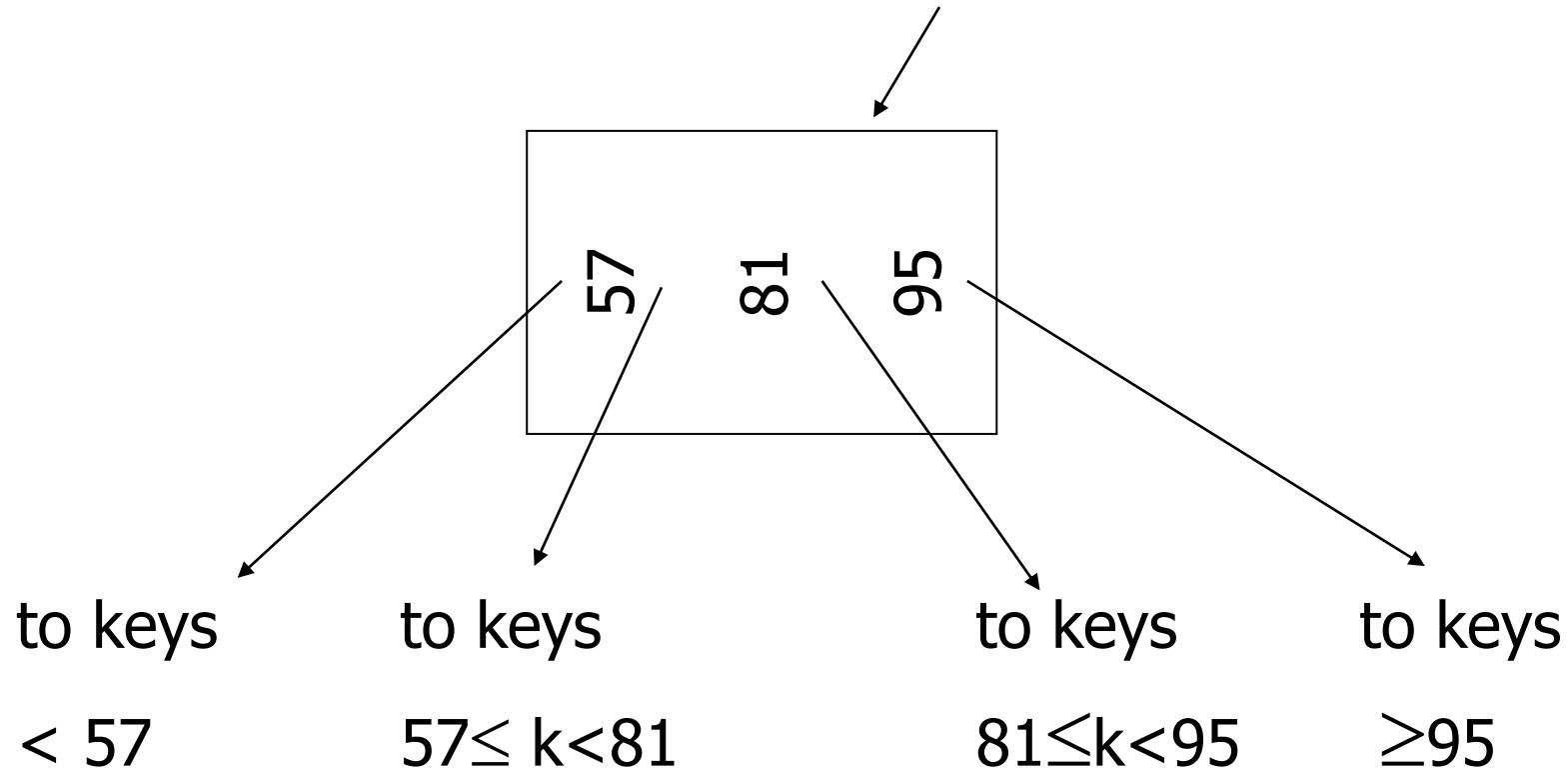
Lookup in a B+ tree

Useful for range queries too

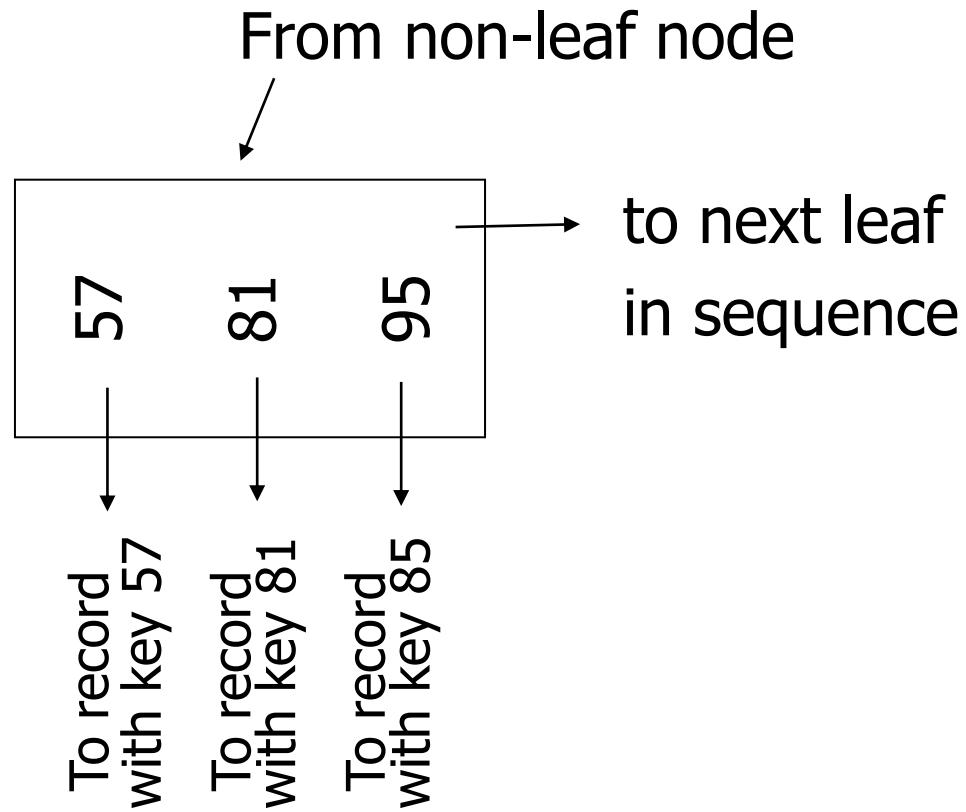
SELECT * FROM R WHERE R.o >= a AND R.o <= b;



Sample non-leaf



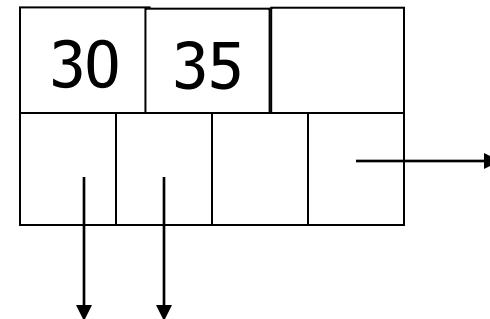
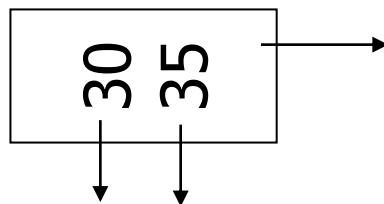
Sample leaf node:



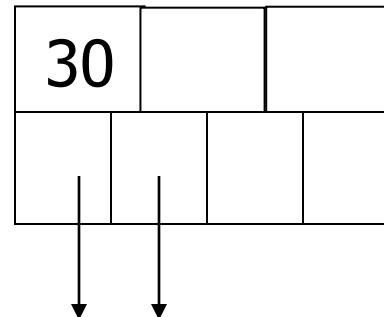
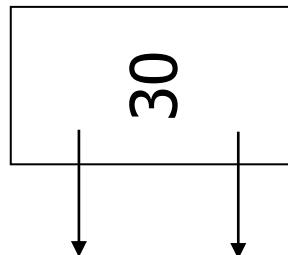
In textbook's notation

$n=3$

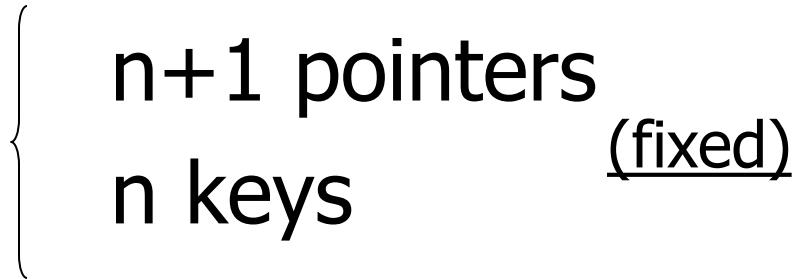
Leaf:



Non-leaf:



Size of nodes:



$n+1$ pointers (fixed)
 n keys

Don't want nodes to be too empty

- Use at least

Non-leaf: $\lceil (n+1)/2 \rceil$ pointers

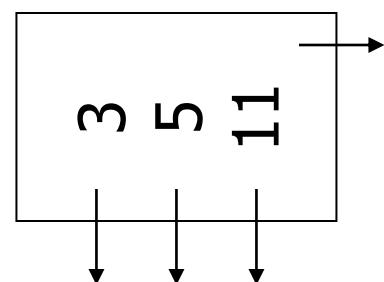
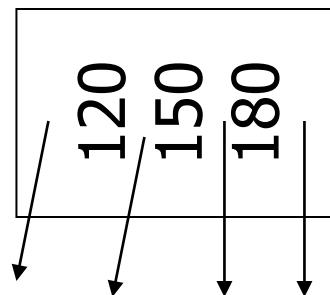
Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to data

n=3

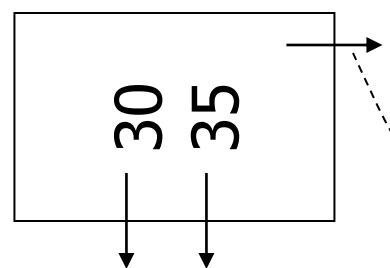
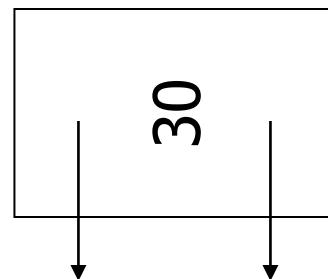
Non-leaf

Leaf

Full node



min. node



counts even if null

B+tree rules tree of order n

- (1) All leaves at same lowest level
(balanced tree)
- (2) Pointers in leaves point to records
except for “sequence pointer”
(to next leaf)

(3) Number of pointers/keys for B+tree

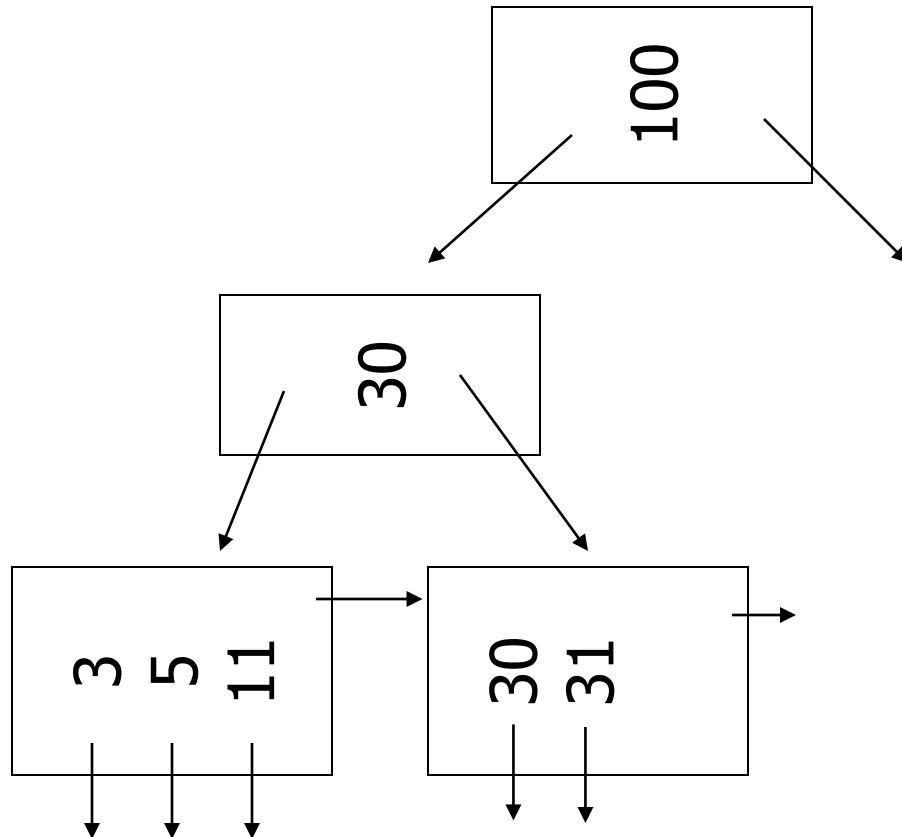
	Max ptrs	Max keys	Min ptrs \rightarrow data	Min keys
Non-leaf (non-root)	$n+1$	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil - 1$
Leaf (non-root)	$n+1$	n	$\lfloor (n+1)/2 \rfloor$	$\lfloor (n+1)/2 \rfloor$
Root	$n+1$	n	1	1

Insert into B+tree

- (a) simple case
 - space available in leaf
- (b) leaf overflow
- (c) non-leaf overflow
- (d) new root

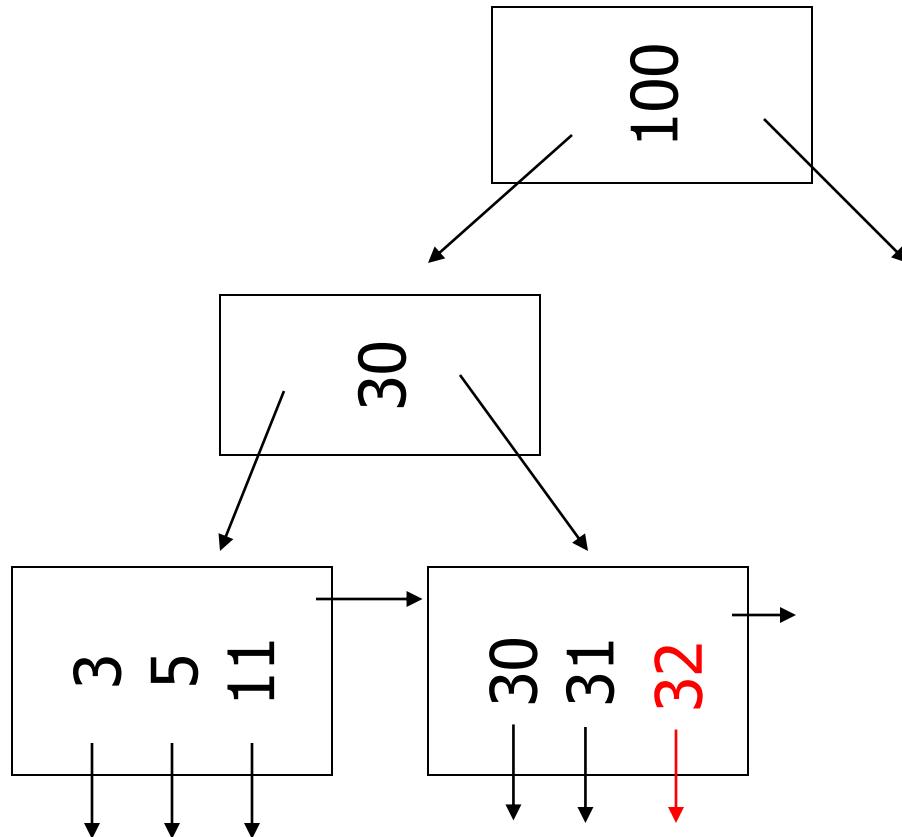
(a) Insert key = 32

n=3



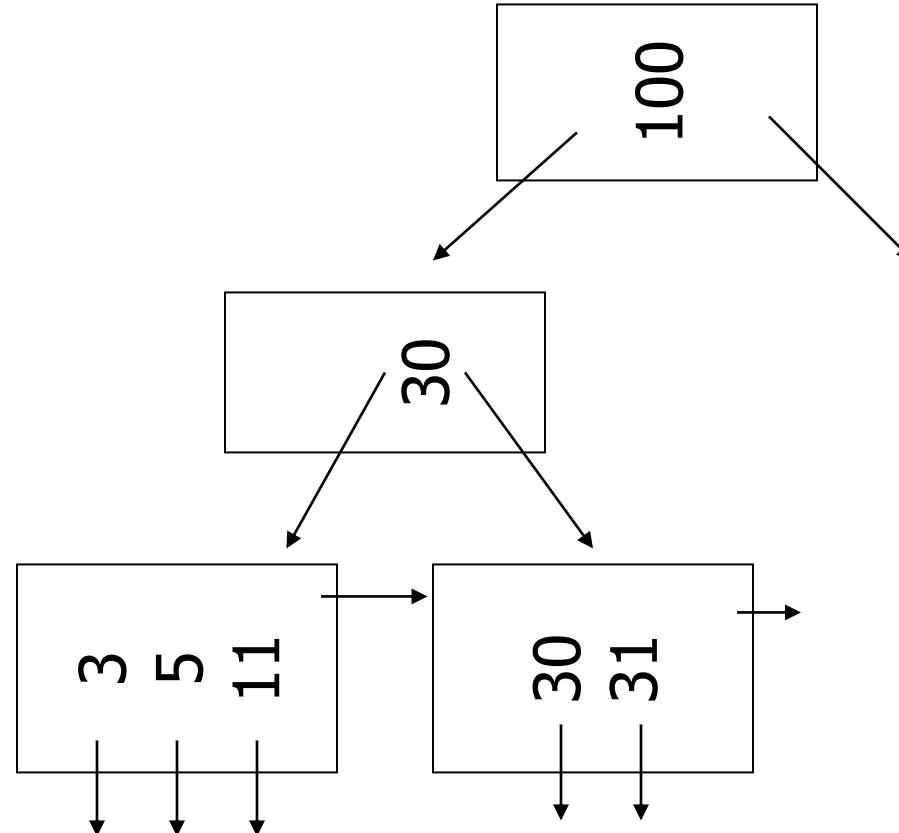
(a) Insert key = 32

n=3



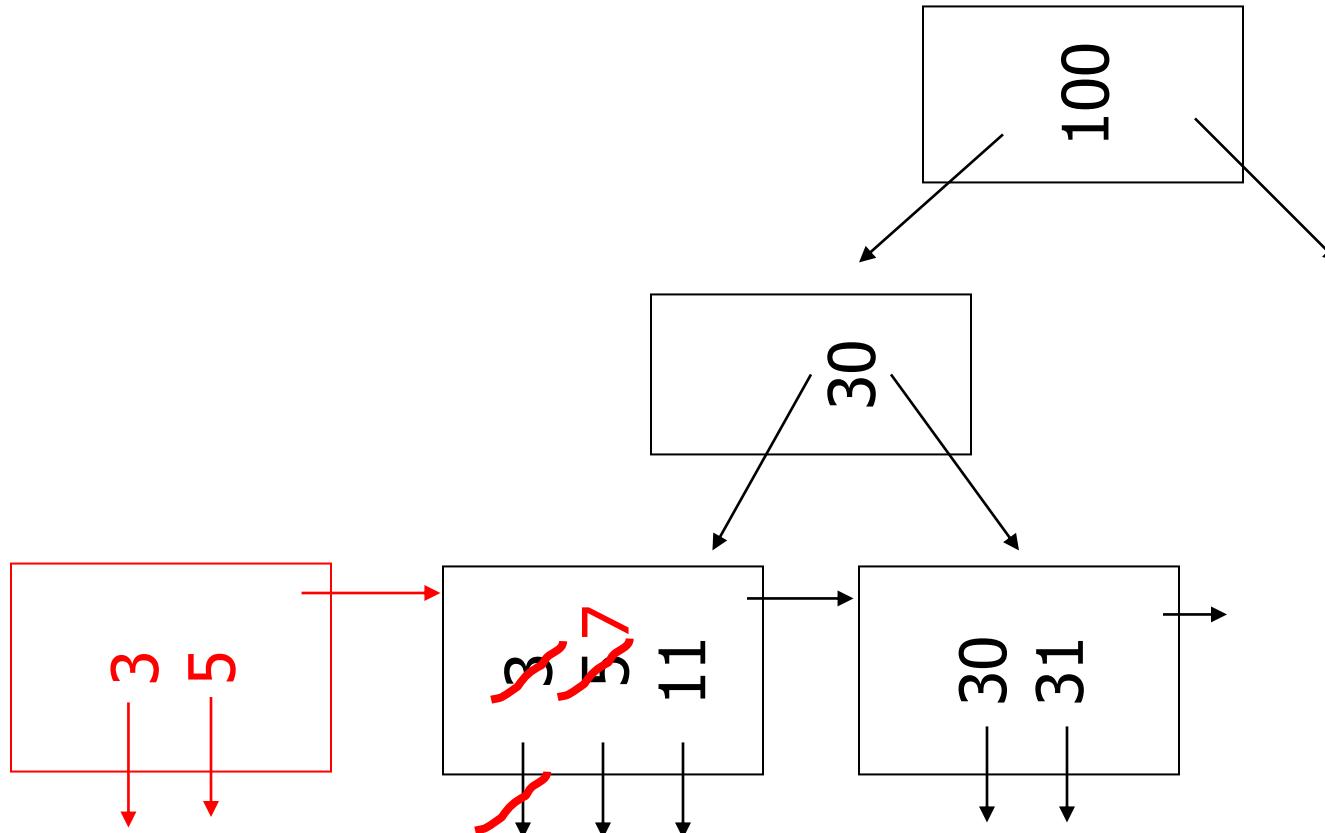
(a) Insert key = 7

n=3



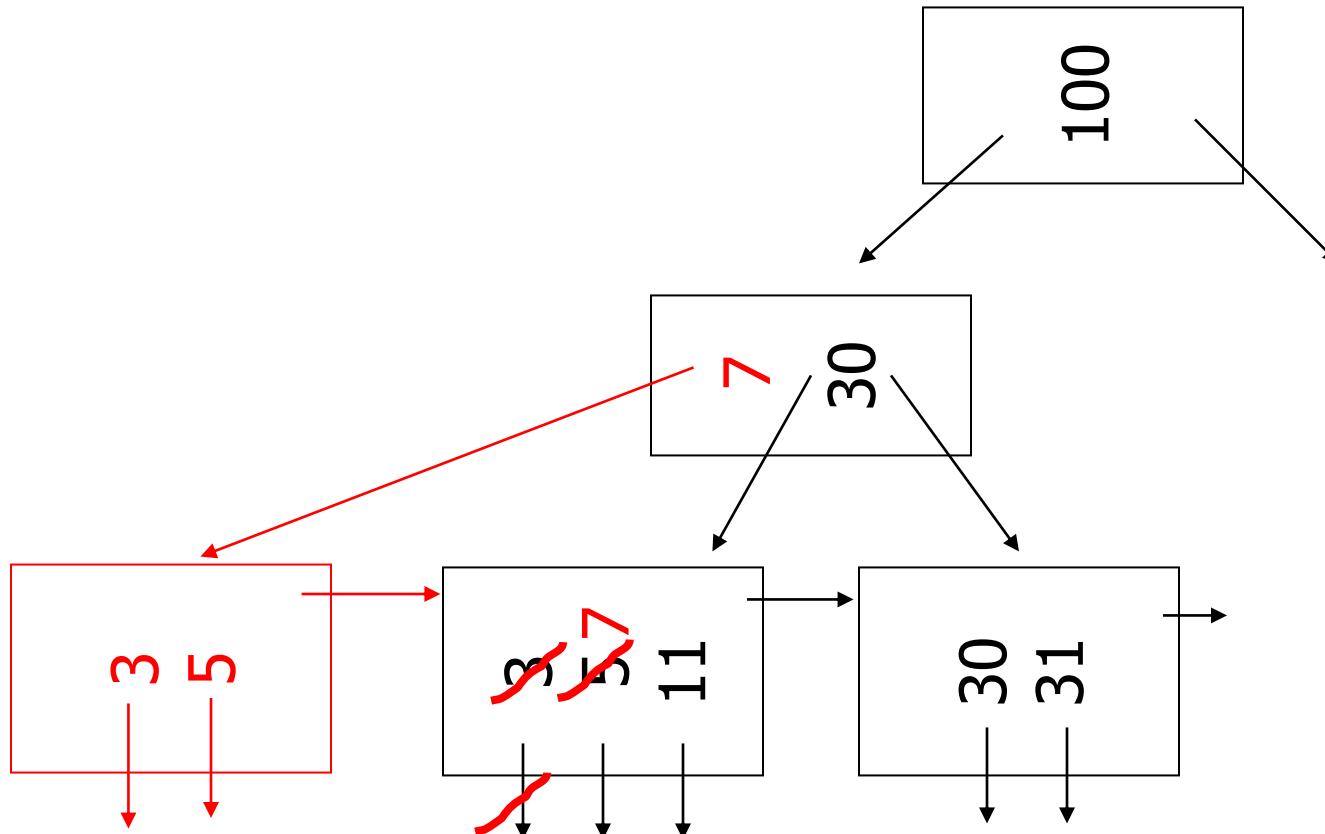
(a) Insert key = 7

n=3



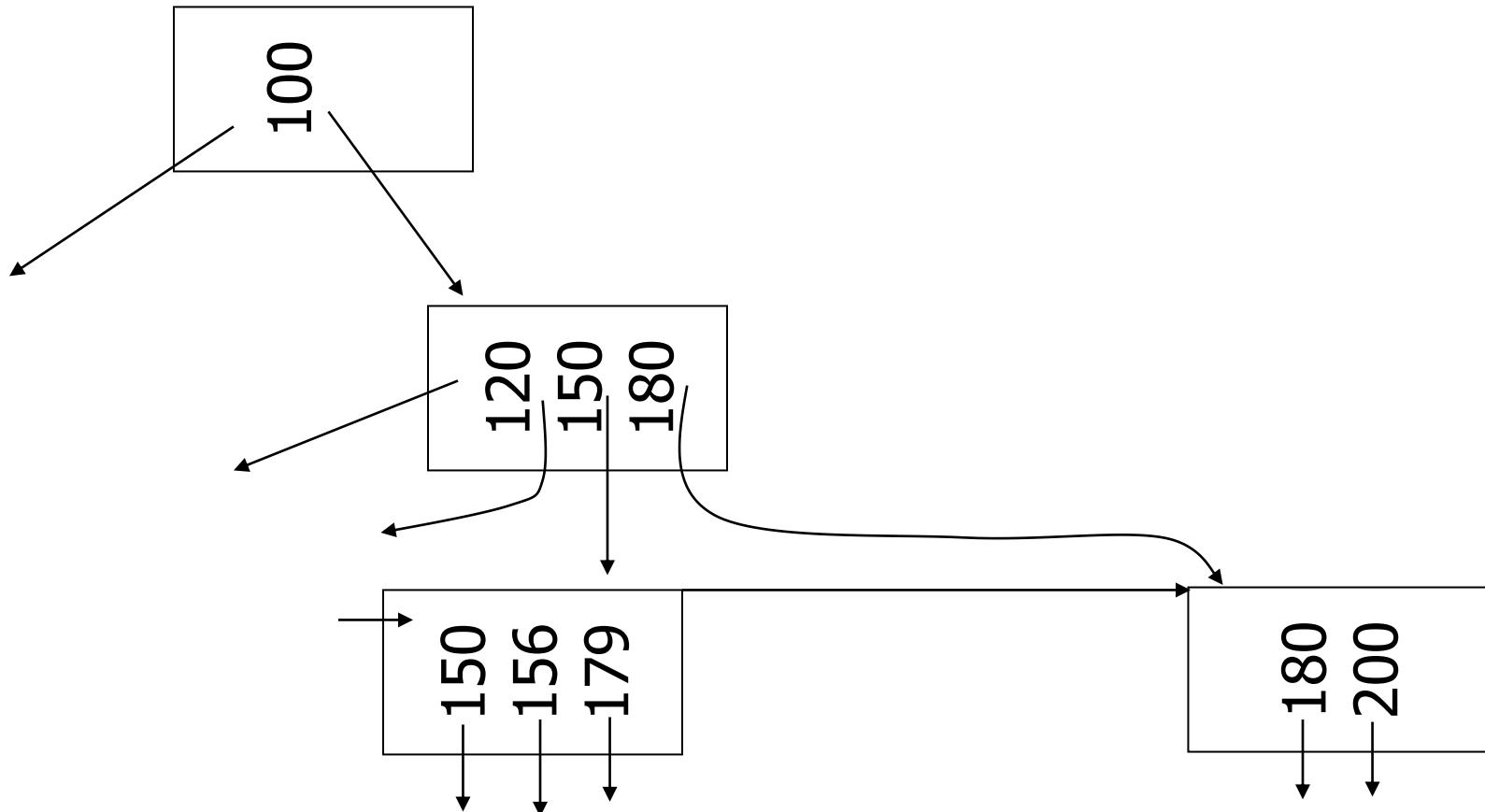
(a) Insert key = 7

n=3



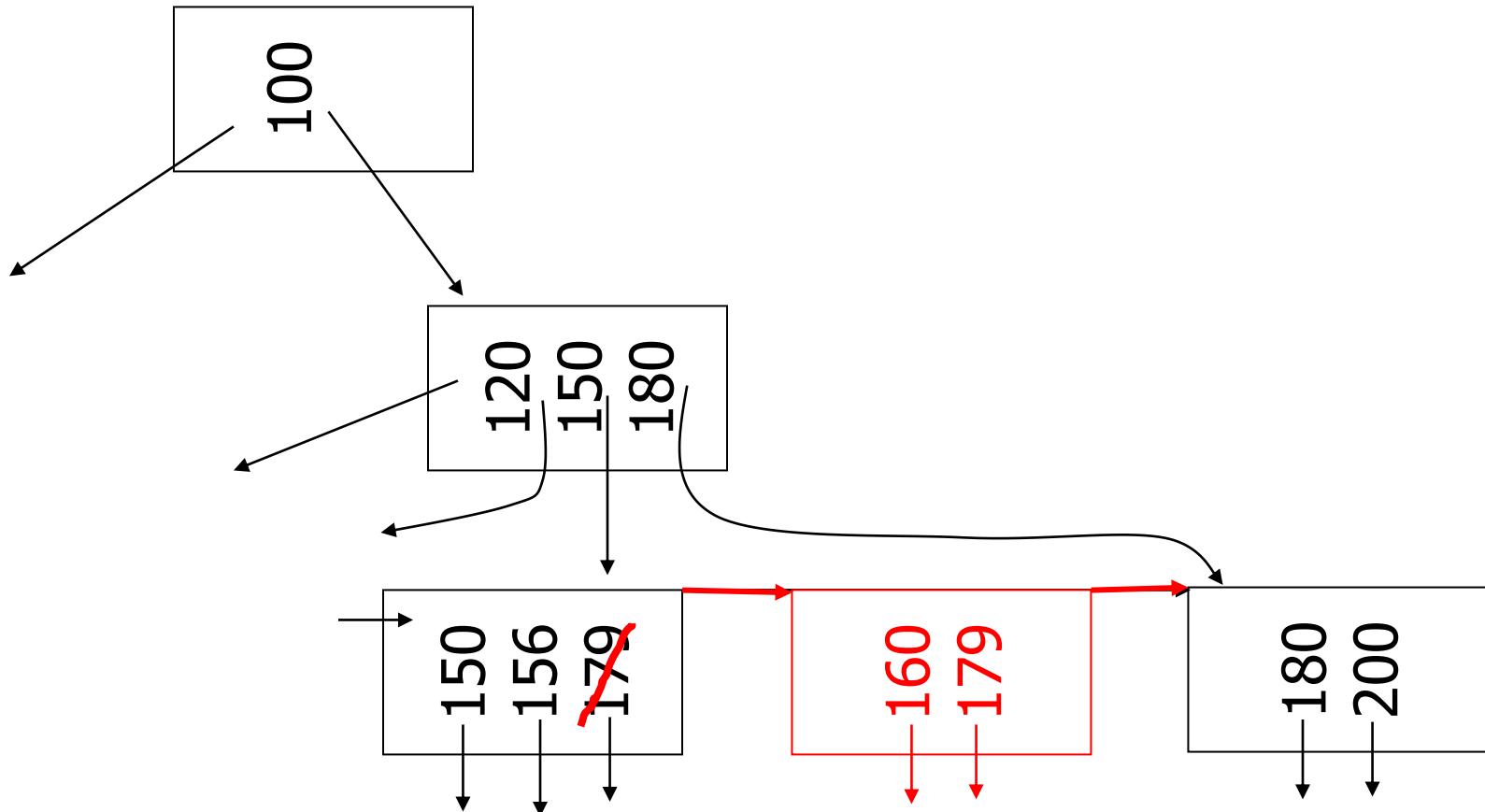
(c) Insert key = 160

n=3



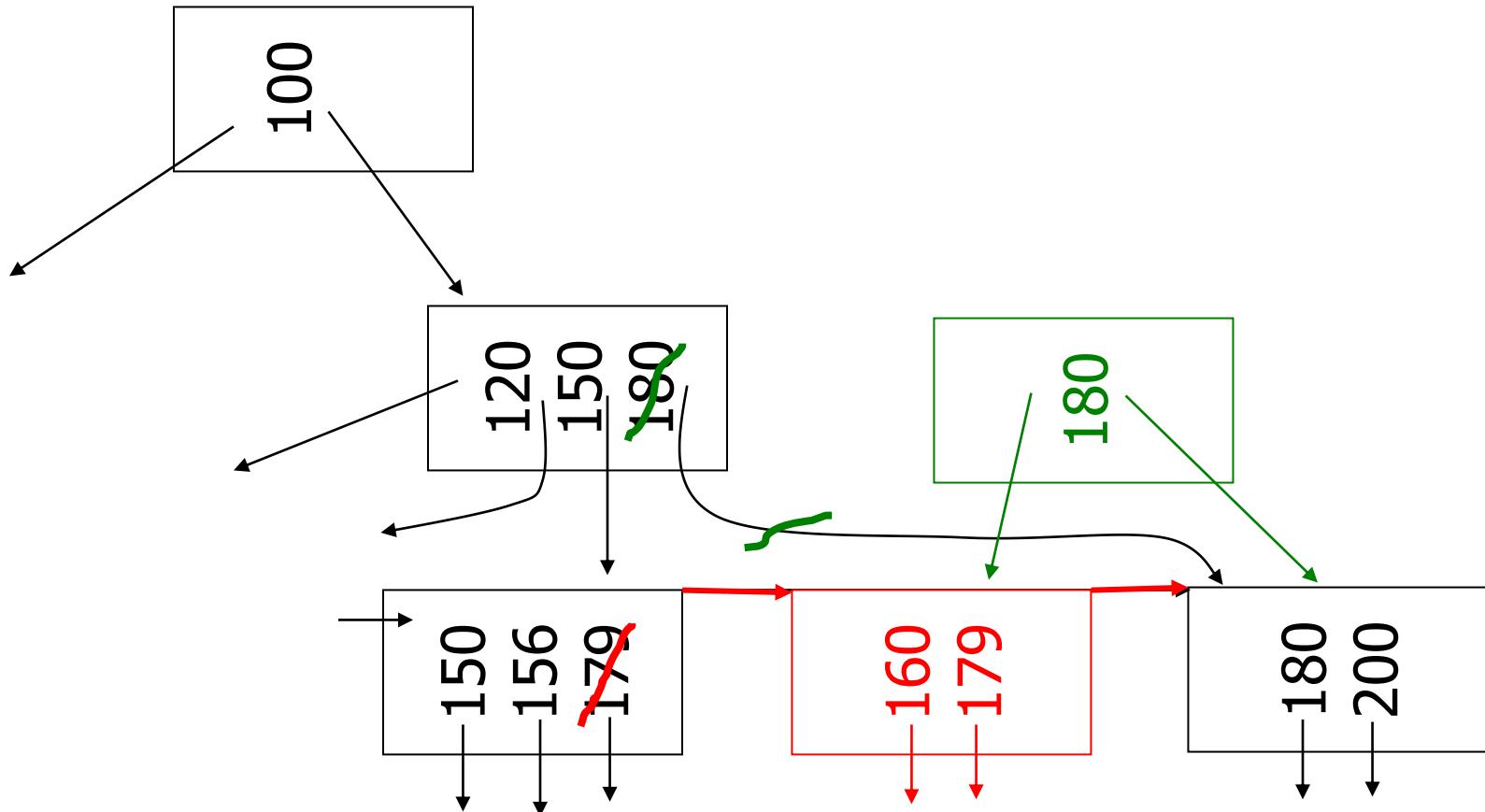
(c) Insert key = 160

n=3



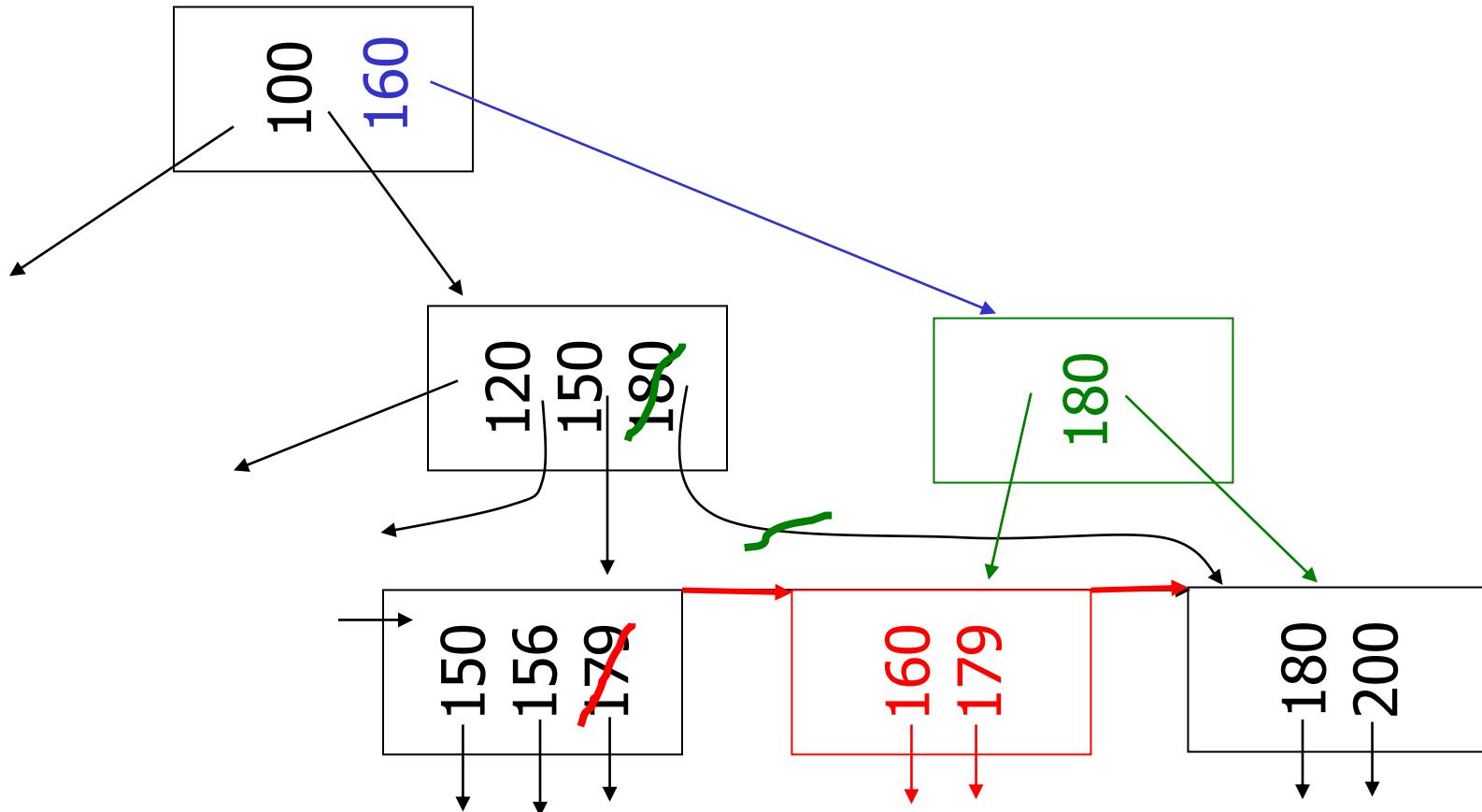
(c) Insert key = 160

n=3



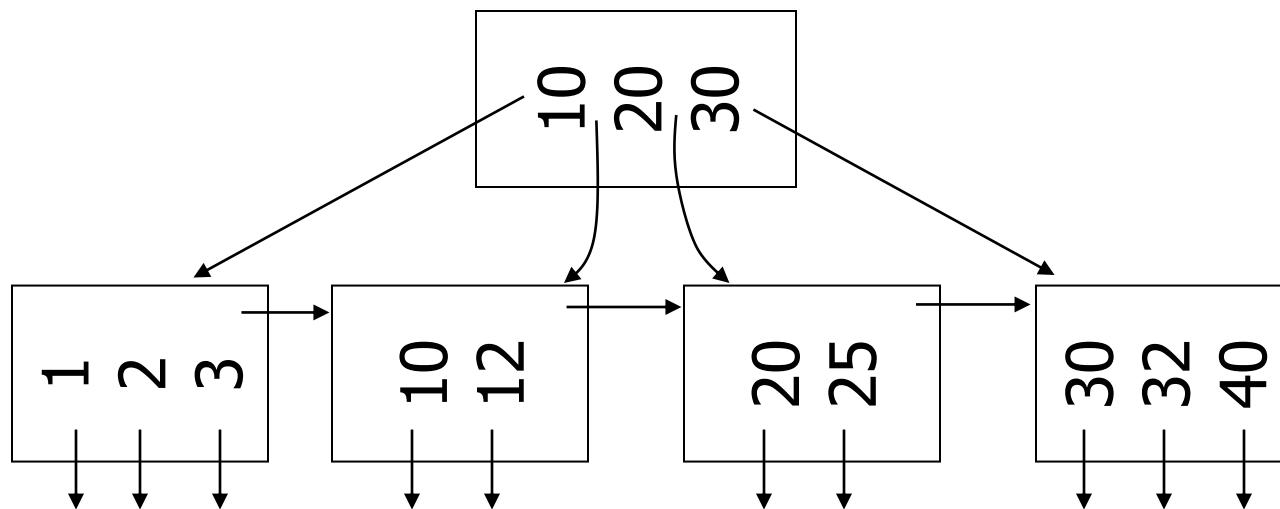
(c) Insert key = 160

n=3



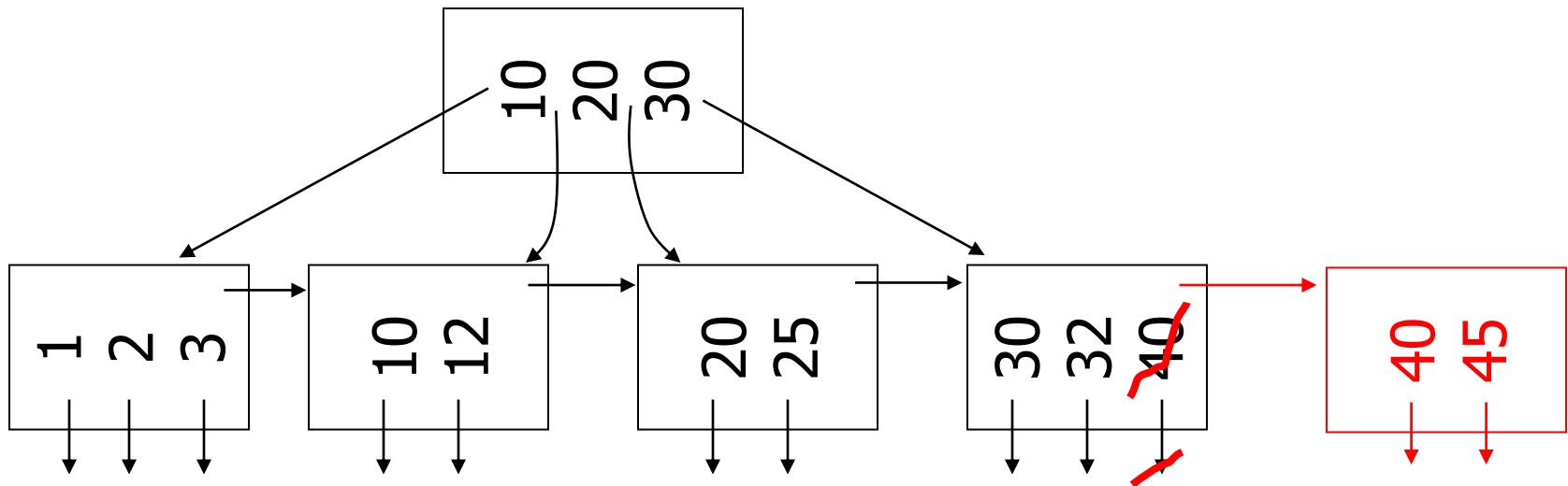
(d) New root, insert 45

n=3



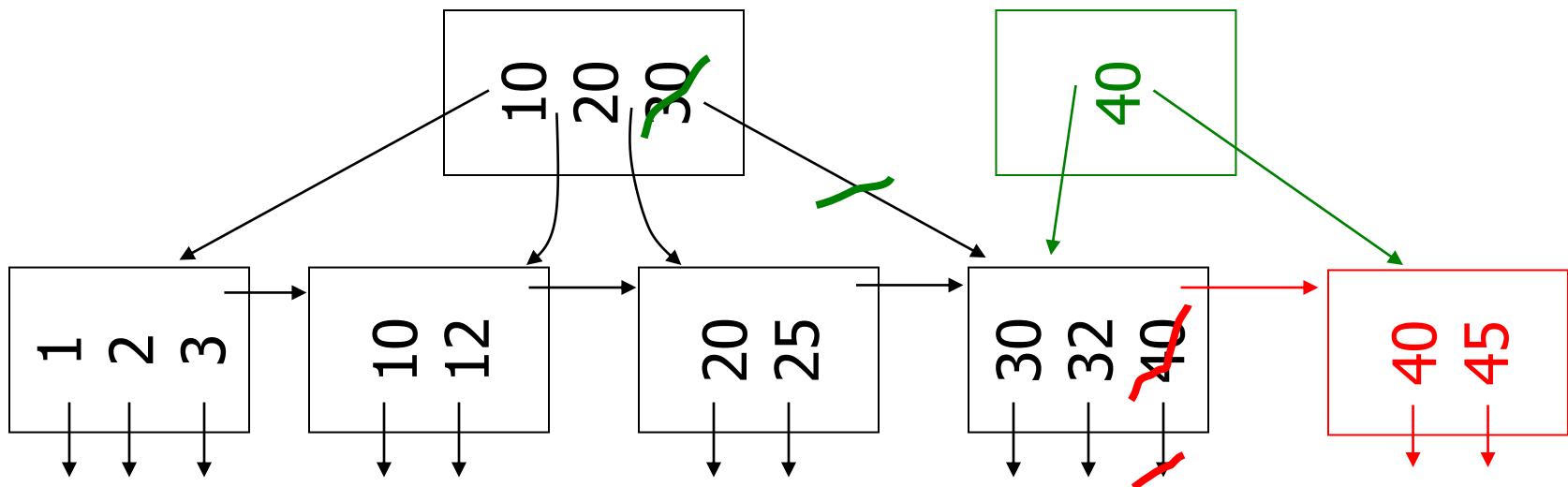
(d) New root, insert 45

n=3



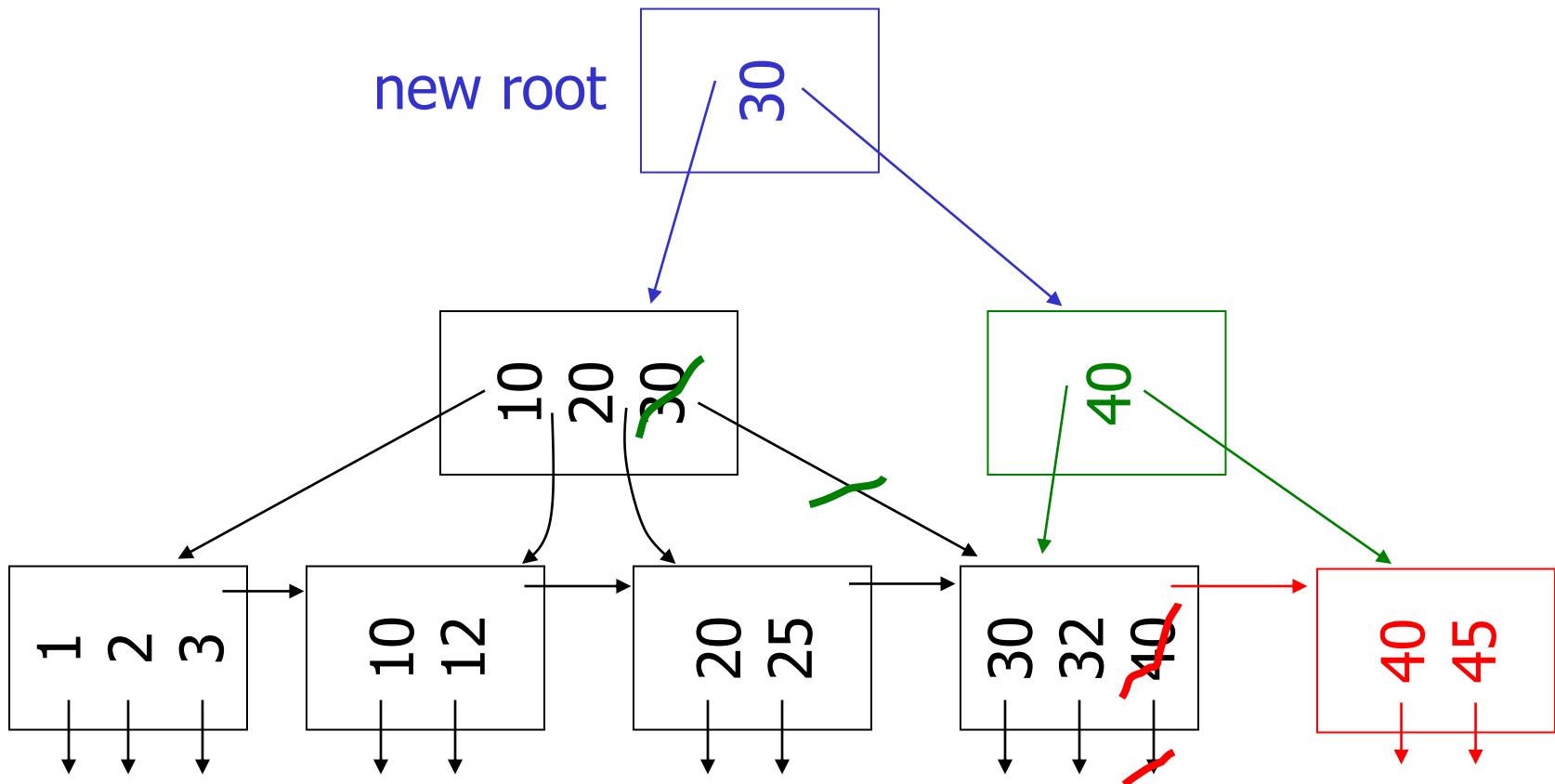
(d) New root, insert 45

n=3



(d) New root, insert 45

n=3



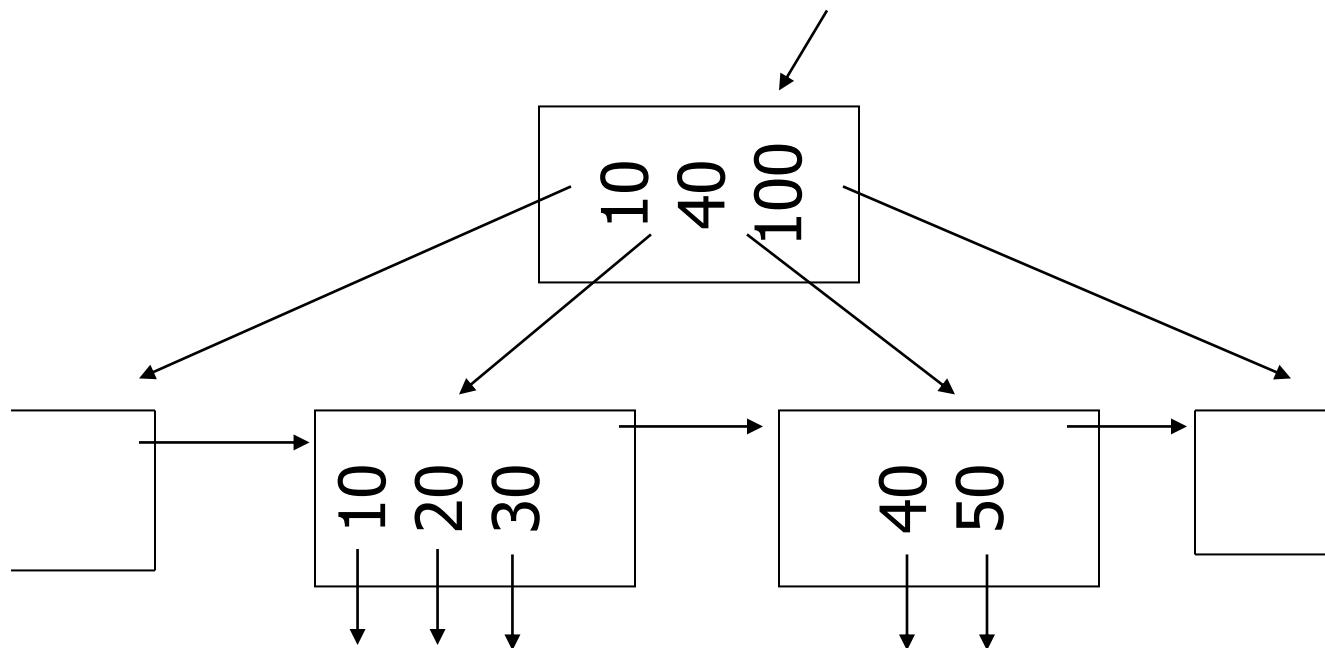
Deletion from B+tree

- (a) Simple case - no example
- (b) Coalesce with neighbor (sibling)
- (c) Re-distribute keys
- (d) Cases (b) or (c) at non-leaf

(b) Coalesce with sibling

– Delete 50

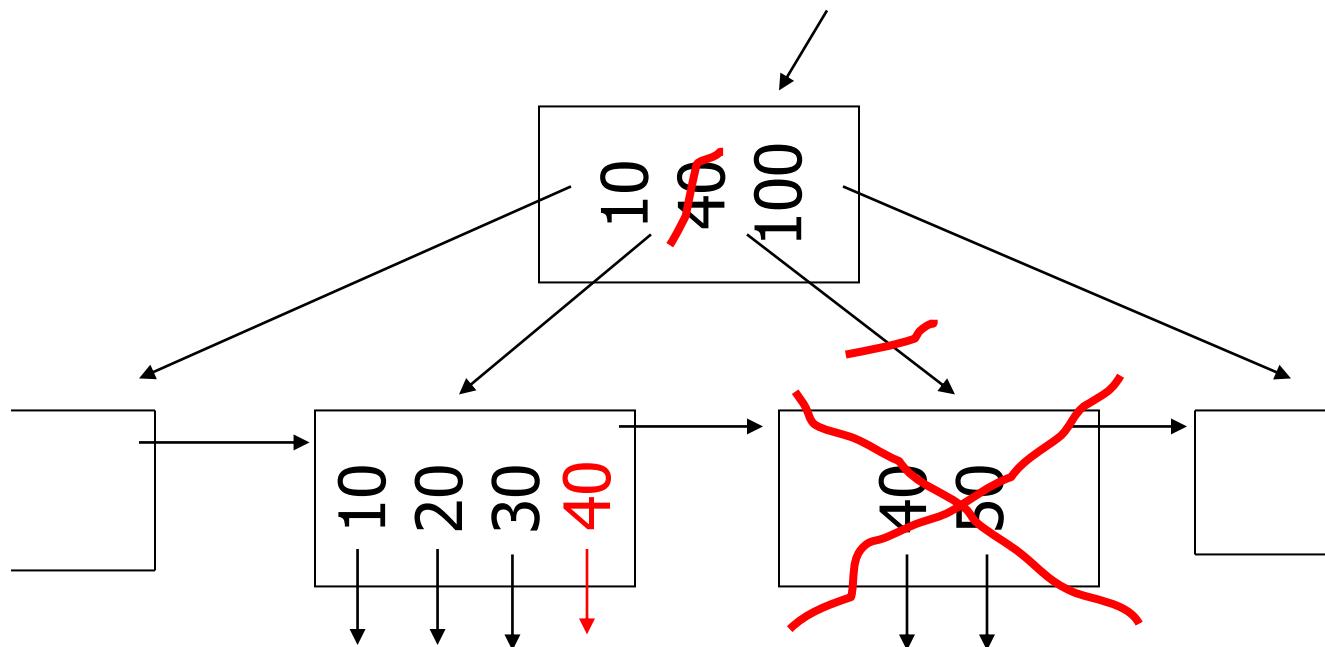
n=4



(b) Coalesce with sibling

– Delete 50

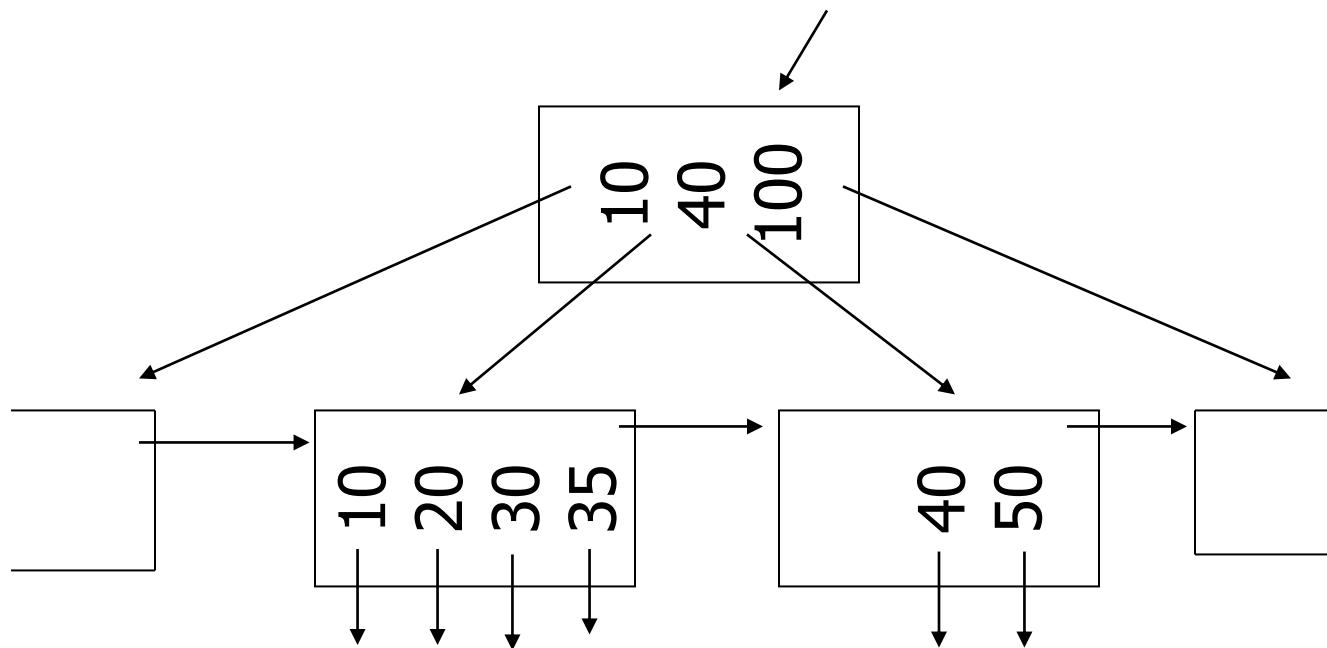
n=4



(c) Redistribute keys

- Delete 50

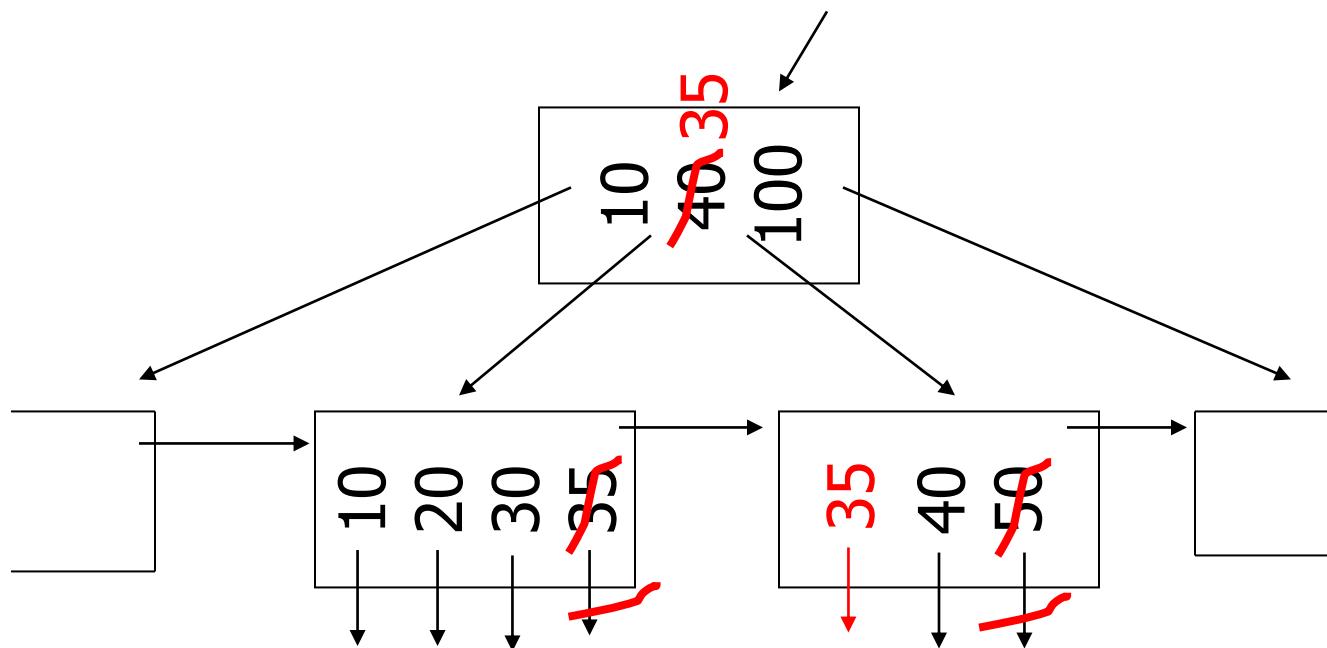
$n=4$



(c) Redistribute keys

– Delete 50

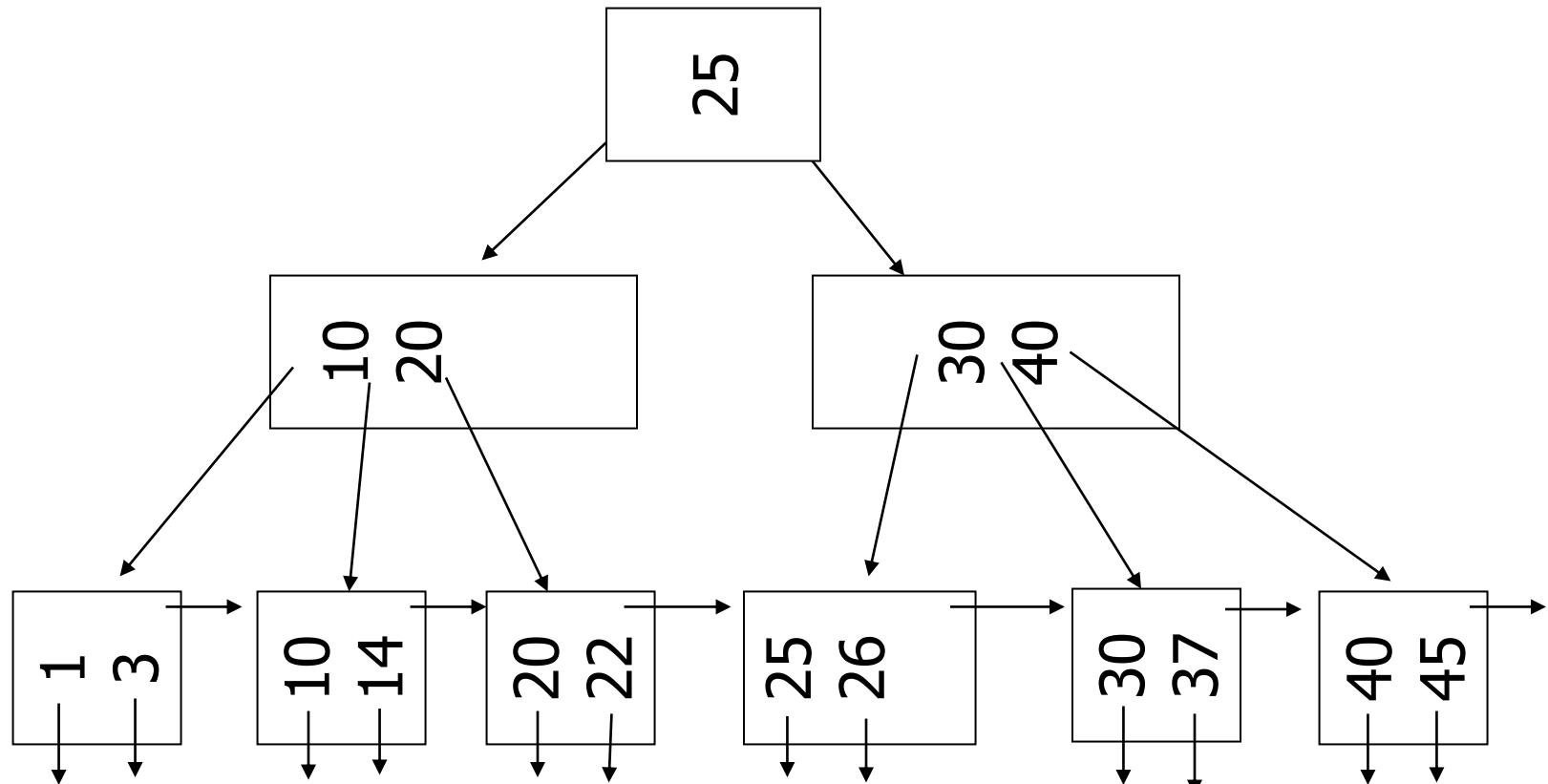
$n=4$



(d) Non-leaf coalesce

– Delete 37

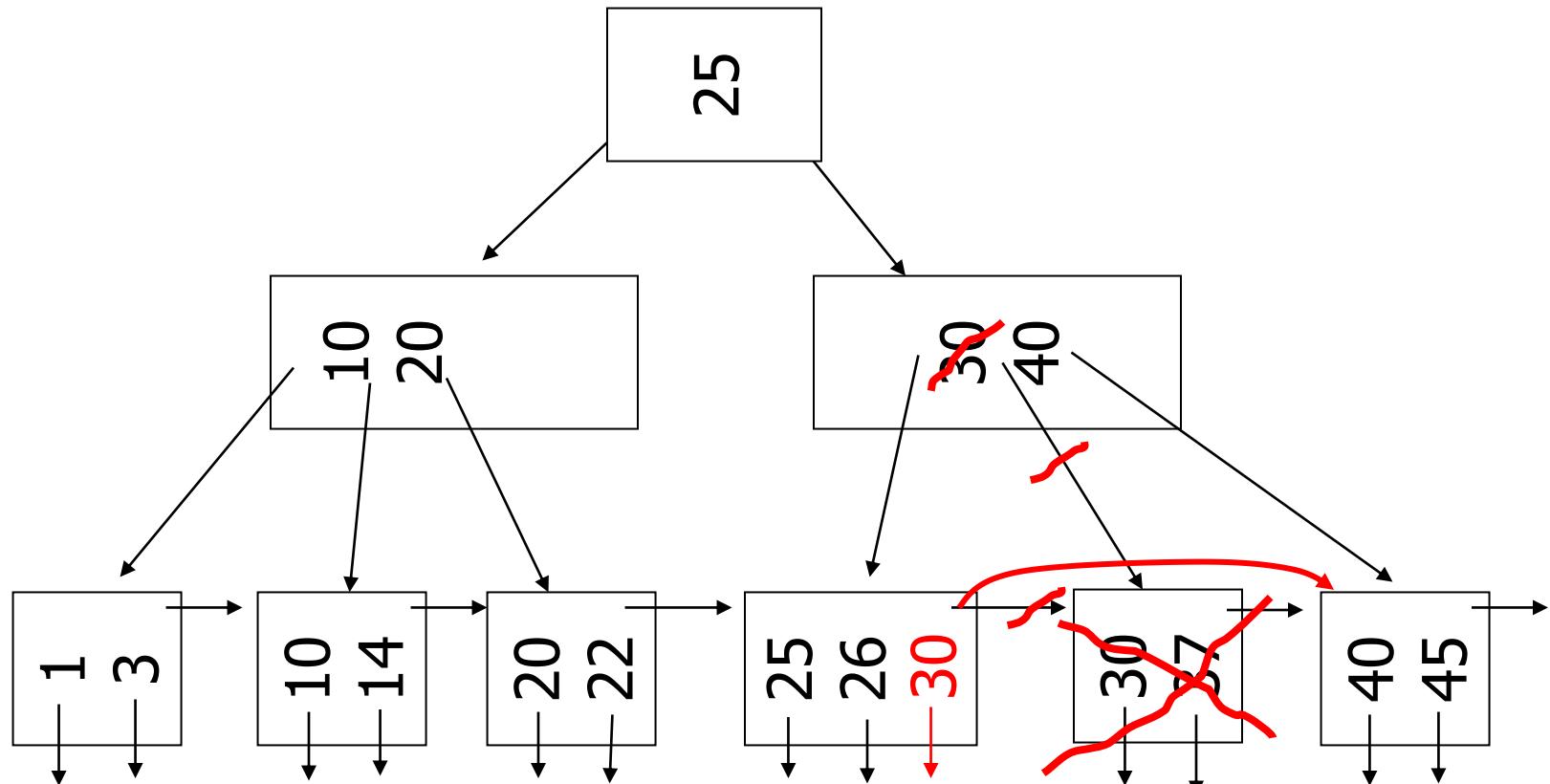
$n=4$



(d) Non-leaf coalesce

– Delete 37

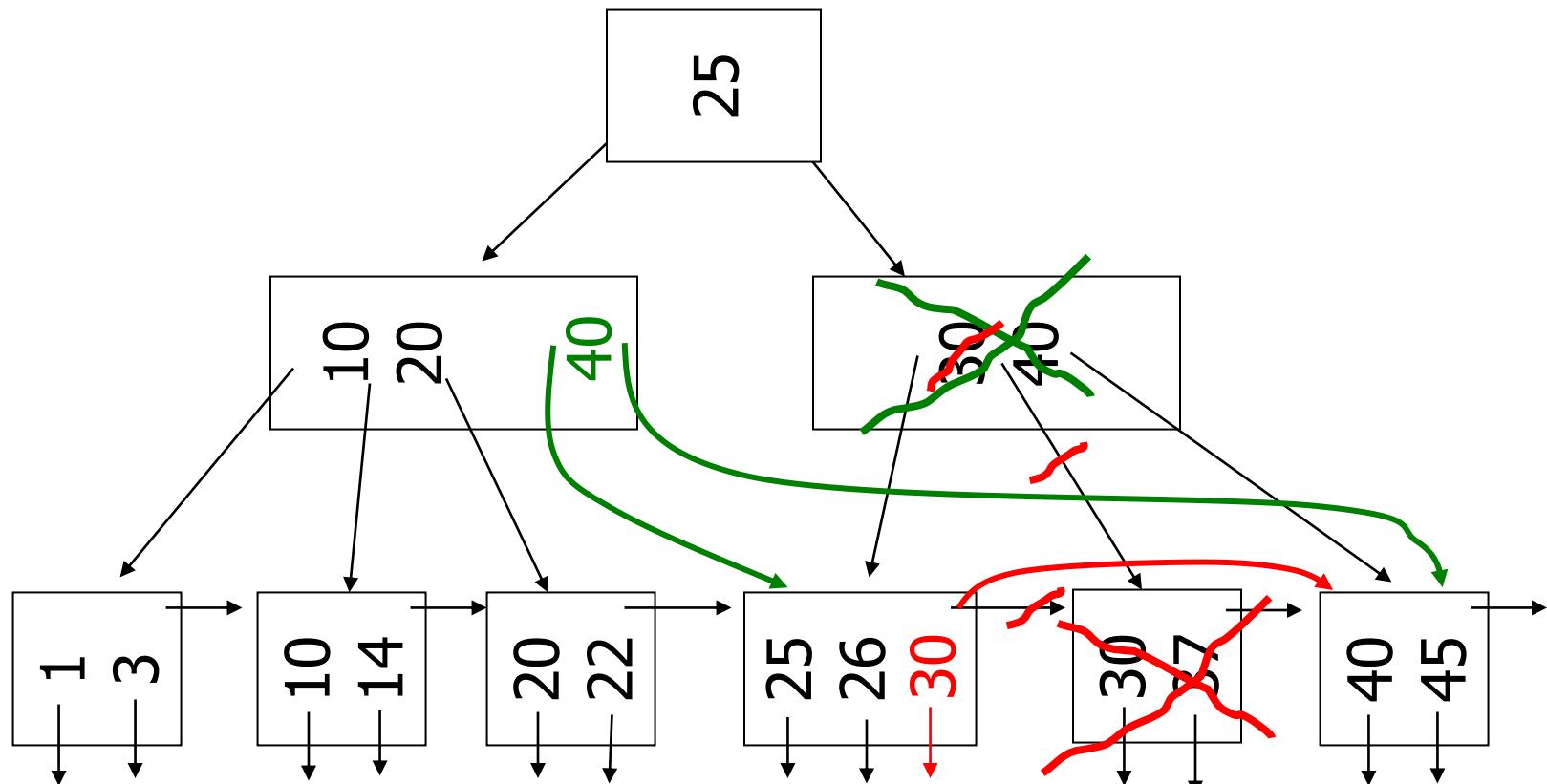
$n=4$



(d) Non-leaf coalesce

– Delete 37

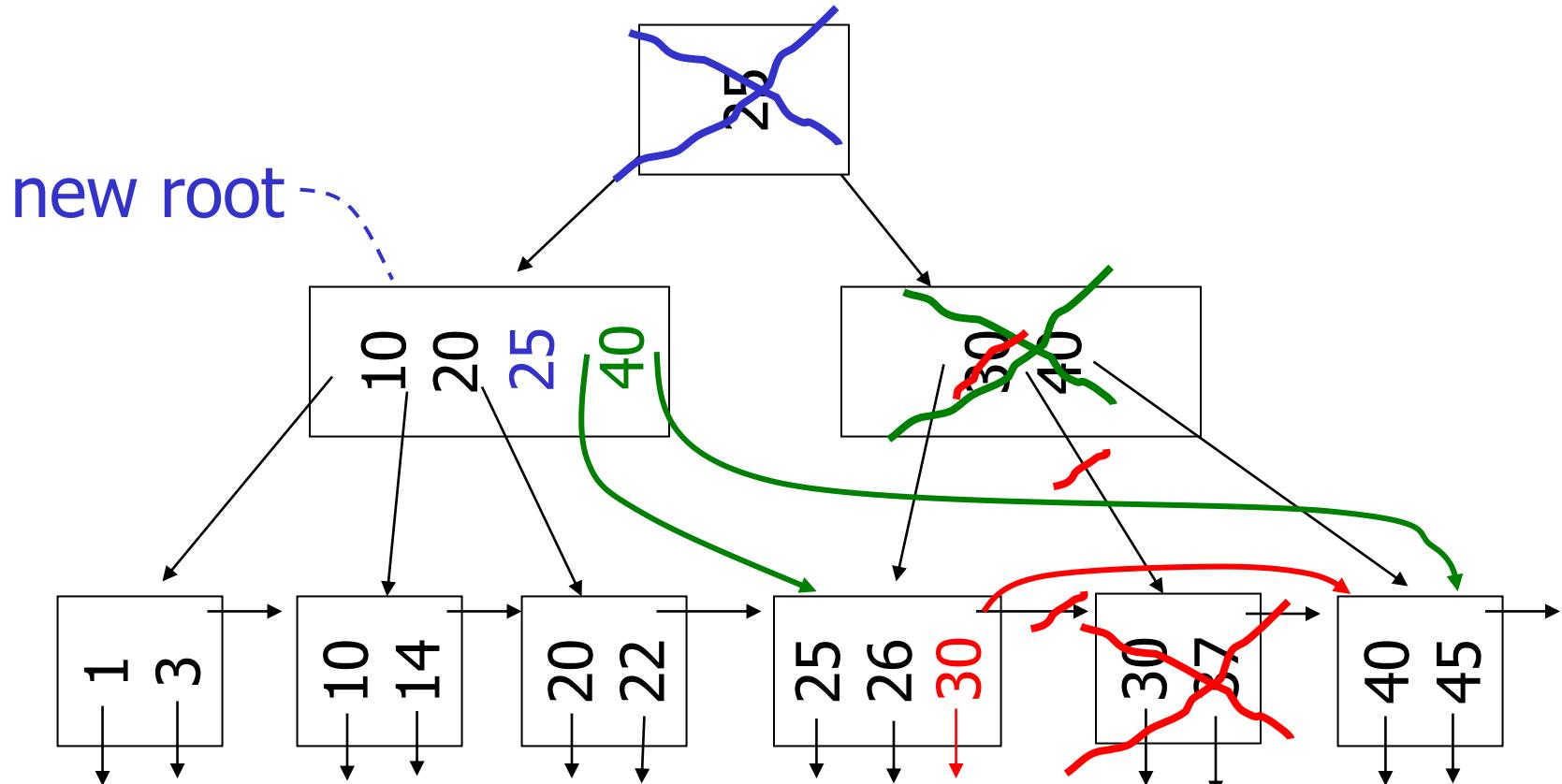
$n=4$



(d) Non-leaf coalesce

n=4

– Delete 37



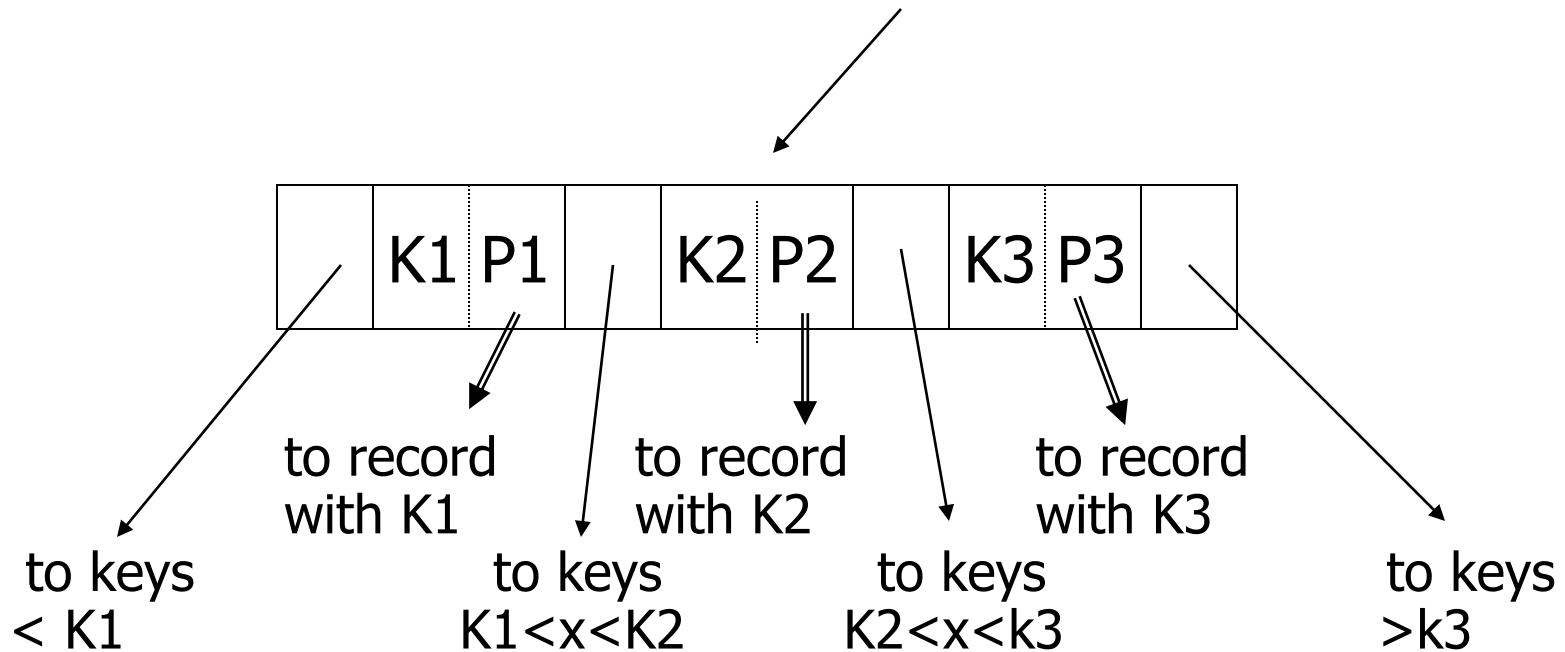
B+tree deletions in practice

- Often, coalescing is not implemented
 - Too hard and not worth it!

- Speaking of buffering...
 - Is LRU a good policy for B+tree buffers?
 - Of course not!
 - Should try to keep root in memory at all times
 - (and perhaps some nodes from second level)

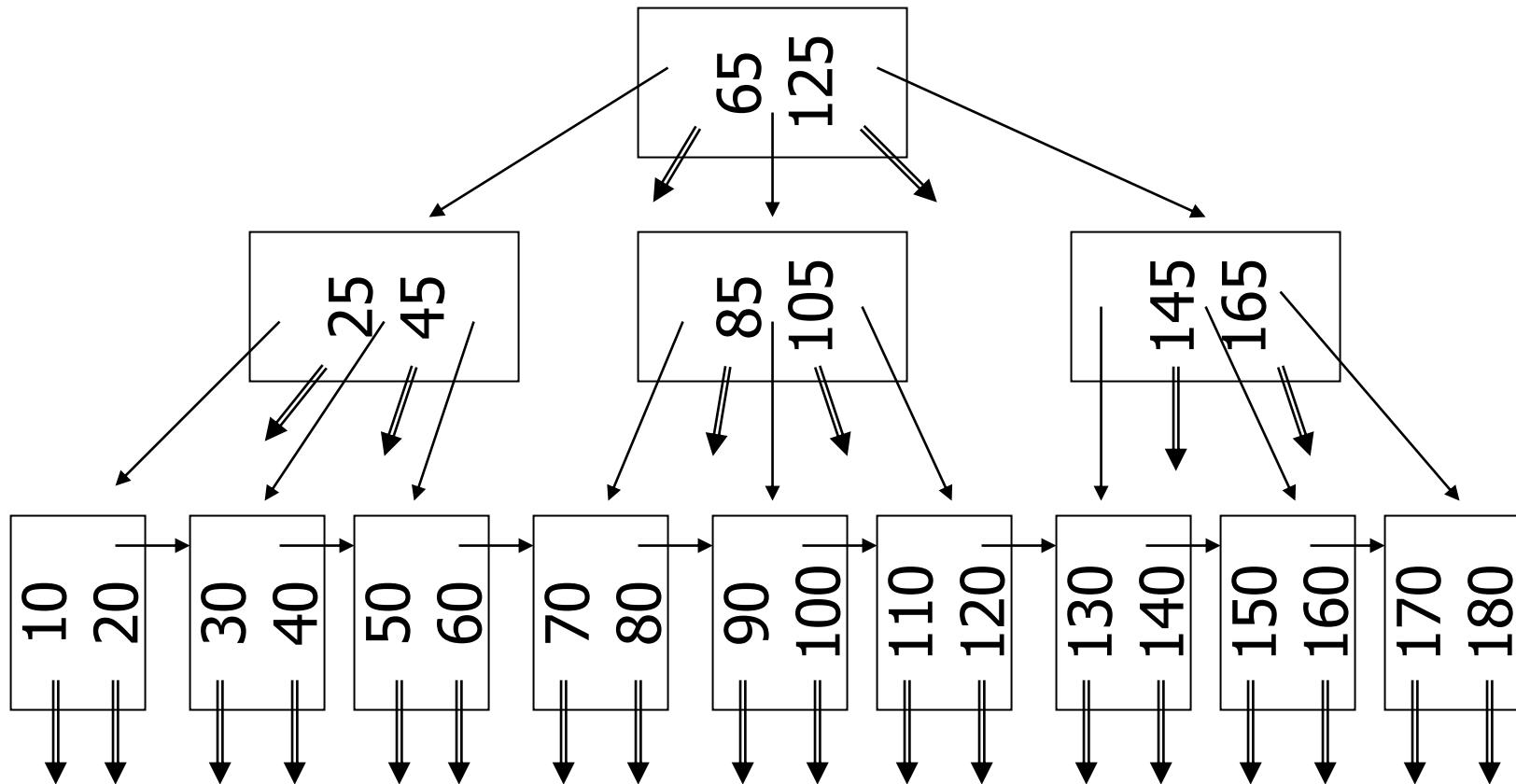
Variation on B+tree: B-tree (no +)

- Idea:
 - Avoid duplicate keys
 - Have record pointers in non-leaf nodes



B-tree example

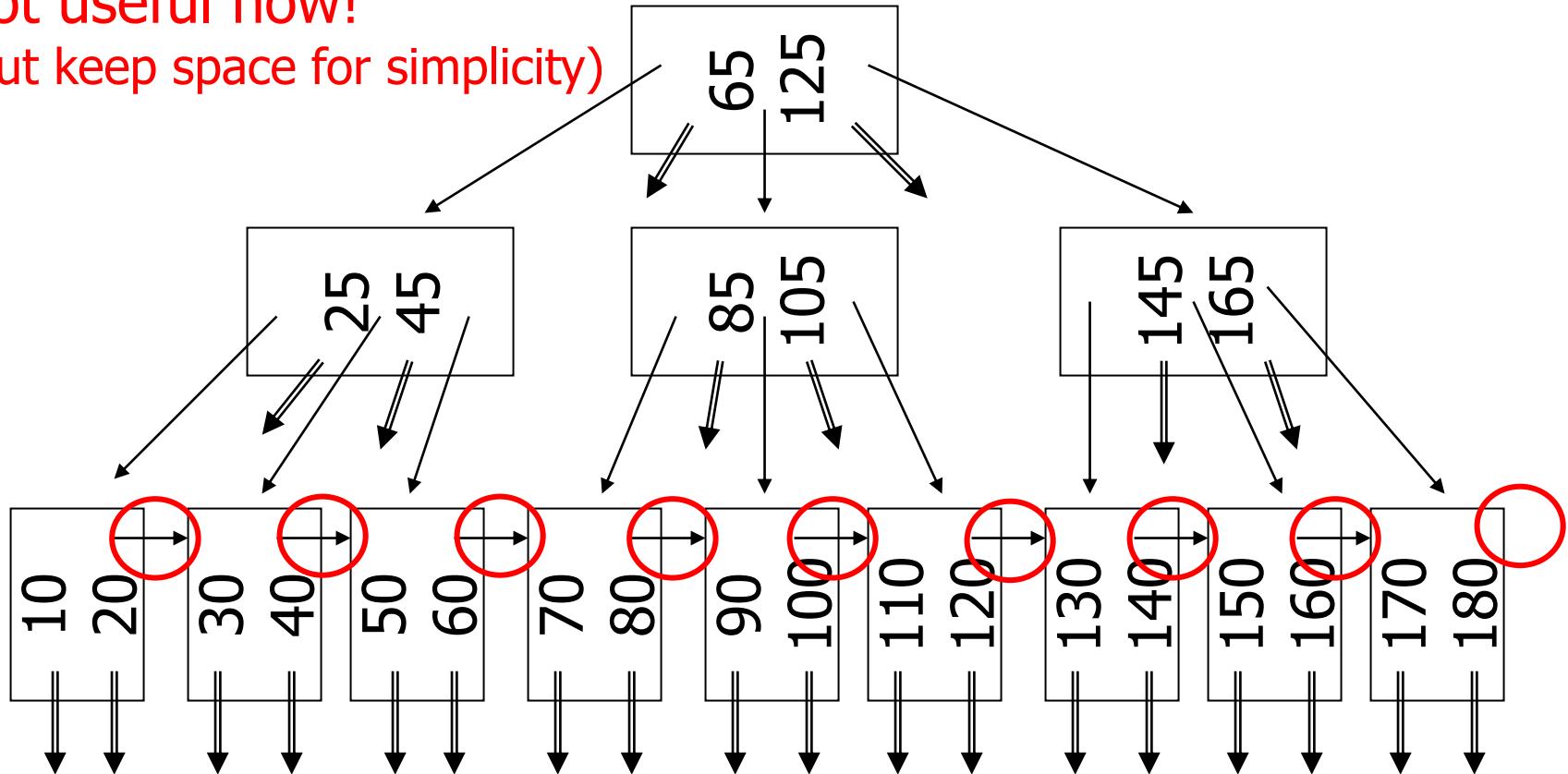
$n=2$



B-tree example

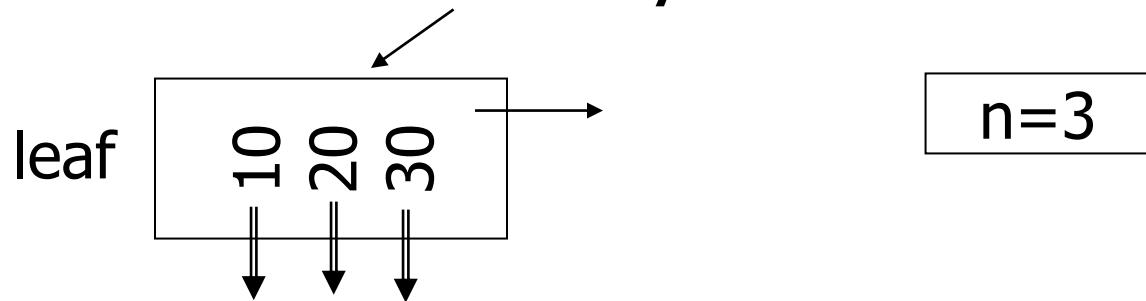
$n=2$

- sequence pointers
not useful now!
(but keep space for simplicity)



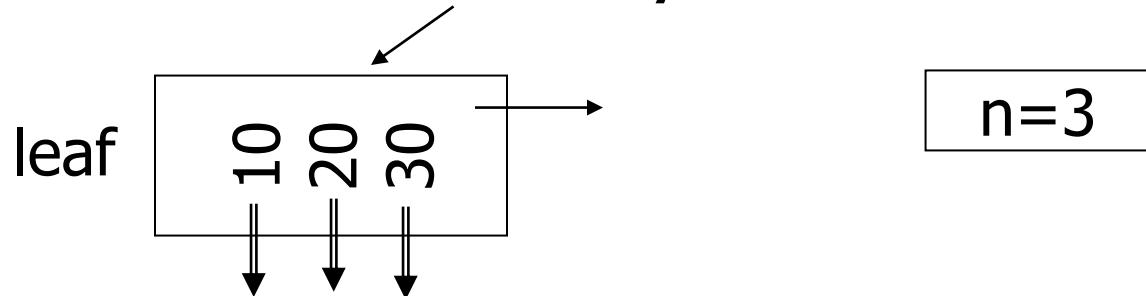
Note on inserts

- Say we insert record with key = 25

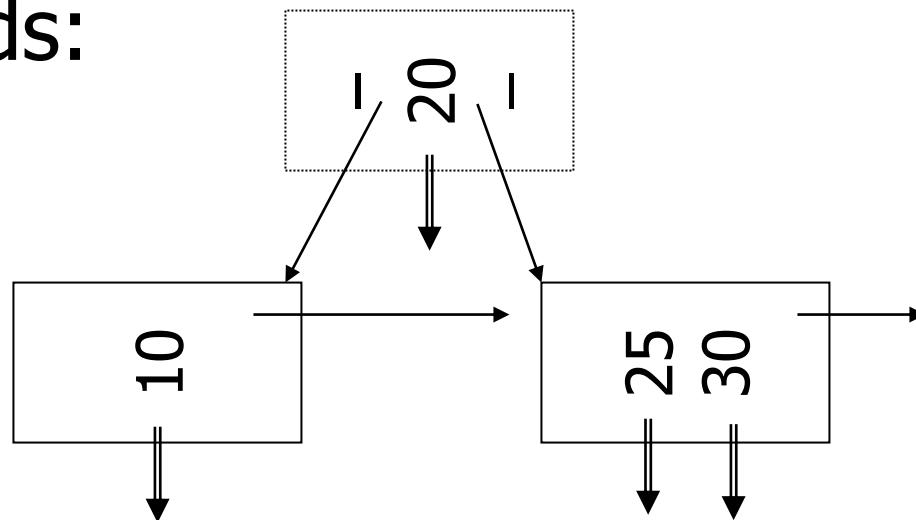


Note on inserts

- Say we insert record with key = 25



- Afterwards:



So, for B-trees:

	MAX			MIN		
	Tree Ptrs	Rec Ptrs	Keys	Tree Ptrs	Rec Ptrs	Keys
Non-leaf non-root	$n+1$	n	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil - 1$	$\lceil (n+1)/2 \rceil - 1$
Leaf non-root	1	n	n	1	$\lfloor n/2 \rfloor$	$\lfloor n/2 \rfloor$
Root non-leaf	$n+1$	n	n	2	1	1
Root Leaf	1	n	n	1	1	1

Tradeoffs:

- 😊 B-trees have faster lookup than B+trees
- 😢 in B-tree, non-leaf & leaf different sizes
- 😢 in B-tree, deletion more complicated

Tradeoffs:

- 😊 B-trees have faster lookup than B+trees
- 😢 in B-tree, non-leaf & leaf different sizes
- 😢 in B-tree, deletion more complicated

→ B+trees preferred!

But note:

- If blocks are fixed size
(due to disk and buffering restrictions)

Then lookup for B+tree is
actually better!!

Outline/summary

- Conventional Indexes
 - Sparse vs. dense
 - Primary vs. secondary
- B trees
 - B+trees vs. B-trees
 - B+trees vs. indexed sequential
- Hashing schemes --> Next

Oracle database concepts

ROWID

Oracle Database uses a **rowid** to uniquely identify a row. Internally, the rowid is a structure that holds information that the database needs **to access a row**. A rowid is not physically stored in the database, but is inferred from the file and block on which the data is stored.

Figure 12-8 ROWID Format

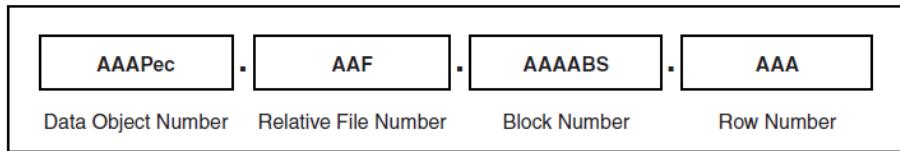
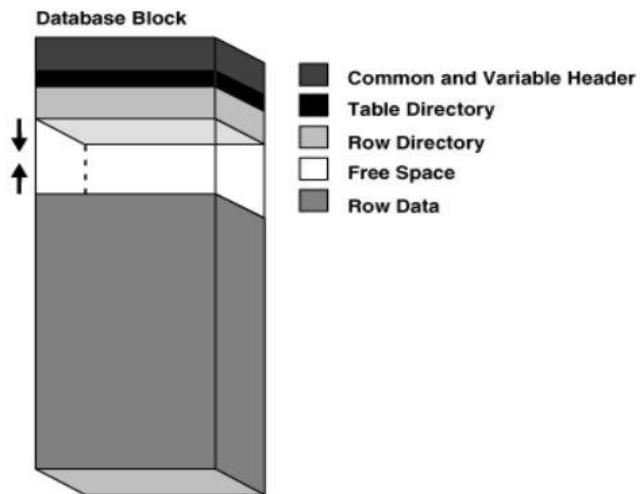


Figure 12-7 Data Block Format



Row directory describes the location of rows in the data portion of the block. The database can place a row anywhere in the bottom of the block. The row address is recorded in one of the slots of the row directory vector. A **rowid** points to a specific file, block, and row number. The row number is **an index** into an entry **in the row directory**. The **row directory entry contains a pointer** to the location of **the row** on the data block. **If the database moves a row within a block**, then the database updates the row directory entry to modify the pointer. The **rowid stays constant**.

A single data segment in a database stores the data for one user object. There are different types of segments. Examples of user segments include:

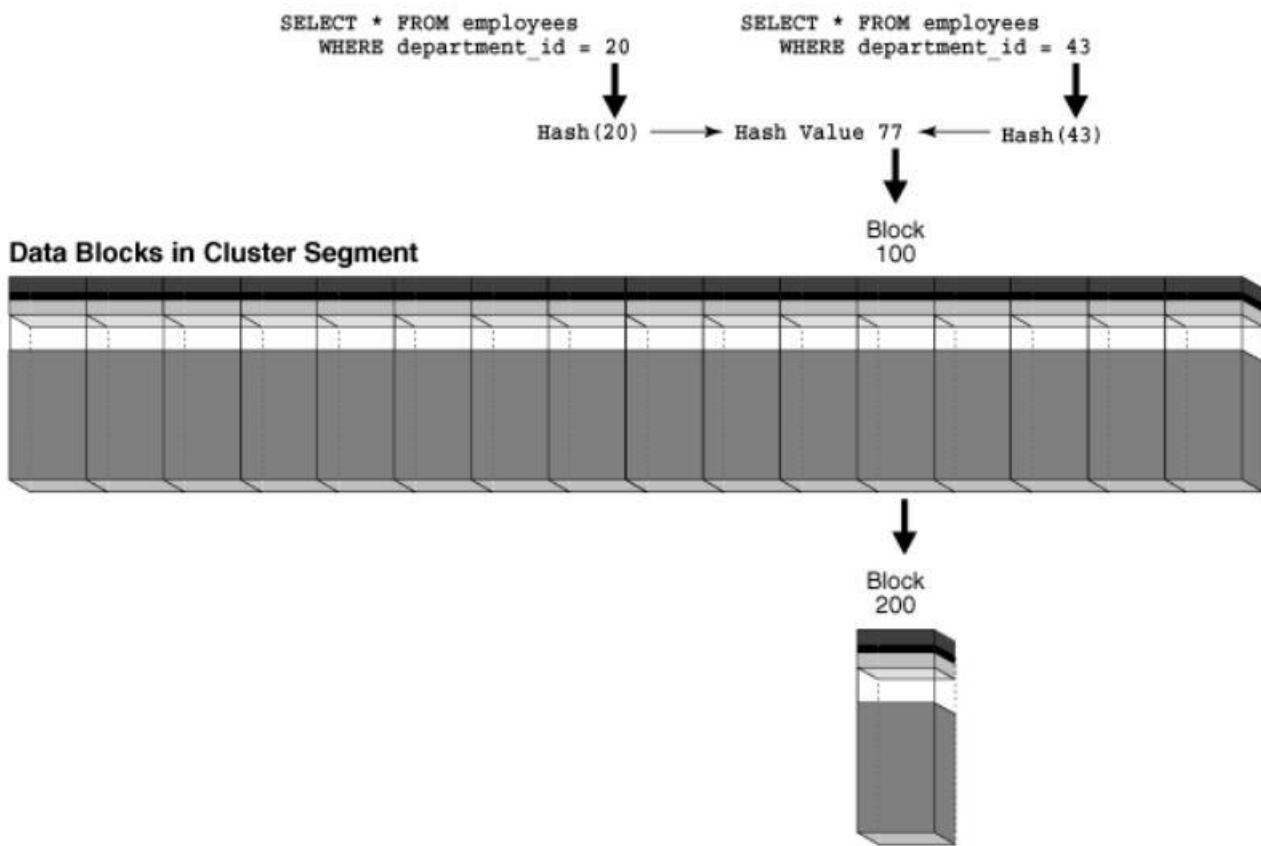
Table, table partition, table cluster, LOB or LOB partition, Index or index partition.

Hash Cluster

To find or store a row in a hash cluster, Oracle Database applies the hash function to the cluster key value of the row. **The resulting hash value corresponds to a data block** in the cluster, which the database reads or writes on behalf of the issued statement.

Hashing multiple input values to the same output value is called a **hash collision**. The database links block 100 to a new **overflow block**, say block 200, and stores the inserted rows in the new block.

Figure 2-7 Retrieving Data from a Hash Cluster When a Hash Collision Occurs



An **index** is an optional structure, associated with a table or table cluster, that can sometimes speed data access. By creating an index on one or more columns of a table, you gain the ability in some cases to retrieve a small set of randomly distributed rows from the table. Indexes are one of many means of **reducing disk I/O**.

A **key** is a set of columns or expressions on which you can build an index. Although the terms are often used interchangeably, indexes and keys are different. Indexes are structures stored in the database that users manage using SQL statements. Keys are strictly a logical concept.

A **composite index**, also called a **concatenated index**, is an **index on multiple columns** in a table.

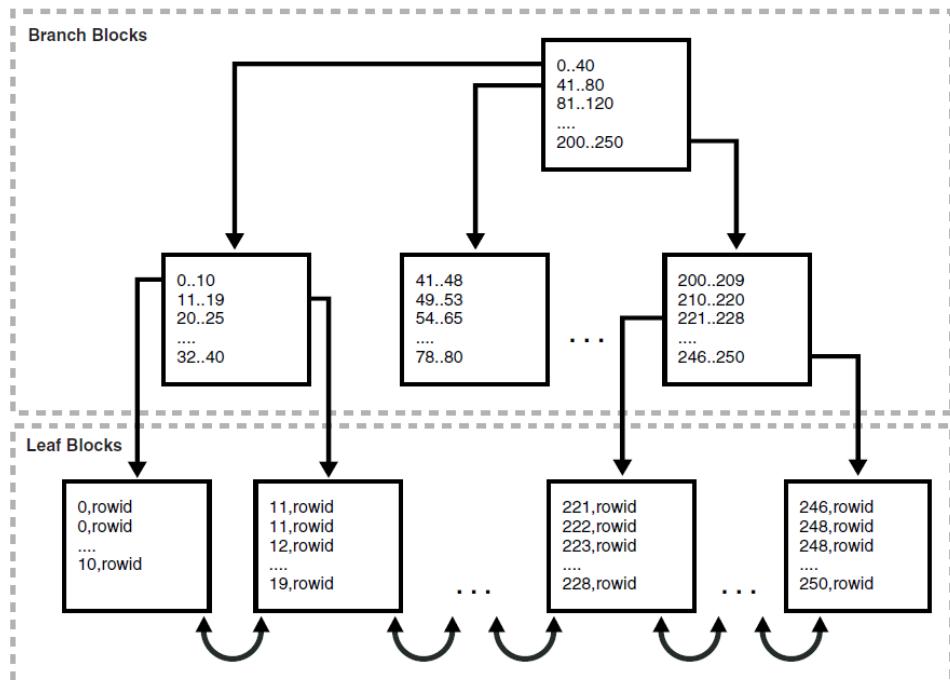
A B-tree index has two types of blocks: **branch blocks** for searching and **leaf blocks** that store values. The upper-level branch blocks of a B-tree index contain index data that points to lower-level index blocks.

Indexes can be **unique** or **nonunique**. Unique indexes guarantee that no two rows of a table have duplicate values in the key column or columns. Nonunique indexes permit duplicates values in the indexed column or columns. Oracle Database does not index table **rows in which all key columns are null**, except for bitmap indexes or when the cluster key column value is null.

In an **ascending** index, Oracle Database stores data in ascending order. By specifying the **DESC** keyword in the `CREATE INDEX` statement, you can create a **descending** index. In this case, the index stores data on a specified column or columns in descending order.

A **reverse key index** is a type of B-tree index that **physically reverses the bytes of each index key** while keeping the column order. For example, if the index key is 20, and if the two bytes stored for this key in hexadecimal are **C1,15** in a standard B-tree index, then a reverse key index stores the bytes as **15,C1**. Reversing the key **solves the problem of contention for leaf blocks** in the right side of a B-tree index.

Figure 3-1 Internal Structure of a B-tree Index



Index Scan

In an index scan, the database retrieves a row by **traversing the index**, using the indexed column values specified by the statement. If the database scans the index for a value, then it will find this value **in n I/Os** where n is the **height** of the B-tree index. This is the basic principle behind Oracle Database indexes.

Full Index Scan

In a full index scan, the database **reads the entire index in order**.

Fast Full Index Scan

A fast full index scan is a full index scan in which the **database accesses the data in the index itself** without accessing the table, and the database **reads the index blocks in no particular order**.

Index Range Scan

An index range scan is an **ordered scan of an index** that has the following characteristics:
One or more leading columns of an index are specified in conditions.

Index Unique Scan

In contrast to an index range scan, an index unique scan must have **either 0 or 1 rowid** associated with an index key. The database performs a unique scan when a predicate references all of the columns in a **UNIQUE** index key using an equality operator. An index unique scan stops processing as soon as it finds the first record because no second record is possible.

Index Skip Scan

An index skip scan **uses logical subindexes** of a composite index. The database "skips" through a single index as if it were searching separate indexes. Skip scanning is beneficial if there are few distinct values in the leading column of a composite index and many distinct values in the nonleading key of the index.

Index Compression

To reduce space in indexes, Oracle Database can employ different compression algorithms. Only keys in the leaf blocks of a B-tree index are compressed. Assume that in a table **repeated values occur** in the first two columns. An index block could have entries as shown in the follow example:

```
online,0,AAAPvCAAFAAAAFAAA  
online,0,AAAPvCAAFAAAAFAAG  
online,0,AAAPvCAAFAAAAFAA1  
online,3,AAAPvCAAFAAAAFAAQ  
online,3,AAAPvCAAFAAAAFAAAT
```

If the index in the preceding example were created with prefix compression **COMPRESS 2**, the index would factor out duplicate occurrences of the first two key values:

```
online,0  
AAAPvCAAFAAAAFAAA  
AAAPvCAAFAAAAFAAG  
AAAPvCAAFAAAAFAA1  
online,3  
AAAPvCAAFAAAAFAAQ  
AAAPvCAAFAAAAFAAAT
```

In a **bitmap index**, the database stores a bitmap for each index key. In a conventional B-tree index, one index entry points to a single row. In a bitmap index, each index key stores pointers to multiple rows. **Each bit in the bitmap corresponds to a possible rowid**. If the bit is set, then the row with the corresponding rowid contains the key value. A mapping function converts the bit position to an actual rowid, so the bitmap index provides the same functionality as a B-tree index although it uses a different internal representation.

The following table illustrates a bitmap index for the gender column of a table:

Value	row1	row2	row3	row4	row5	row6	row7
M	1	0	1	1	1	0	0
F	0	1	0	0	0	1	1

Function-based Index

You can create indexes **on functions and expressions** that involve one or more columns in the table being indexed. A function-based index computes the value of a function or expression involving one or more columns and stores it in the index. A function-based index can be either a **B-tree or a bitmap index**. Function-based indexes are efficient for evaluating statements that contain functions in their WHERE clauses.

Index-Organized Table

An index-organized table is a table stored in a variation of a B-tree index structure. In a heap-organized table, rows are inserted where they fit. In an index-organized table, rows are stored in an

index defined on the primary key for the table. Each index entry in the B-tree also stores the non-key column values.

An index-organized table stores all data in the same structure and does not need to store the rowid.

When creating an index-organized table, you can specify a separate segment as a row overflow area. In index-organized tables, B-tree index entries can be large because they contain an entire row, so a separate segment to contain the entries is useful. In contrast, B-tree entries are usually small because they consist of the key and rowid.

If a row overflow area is specified, then the database can divide a row in an index-organized table into the following parts:

The index entry

This part contains column values for all the primary key columns, a physical rowid that points to the overflow part of the row, and optionally a few of the non-key columns. This part is stored in the index segment.

The overflow part

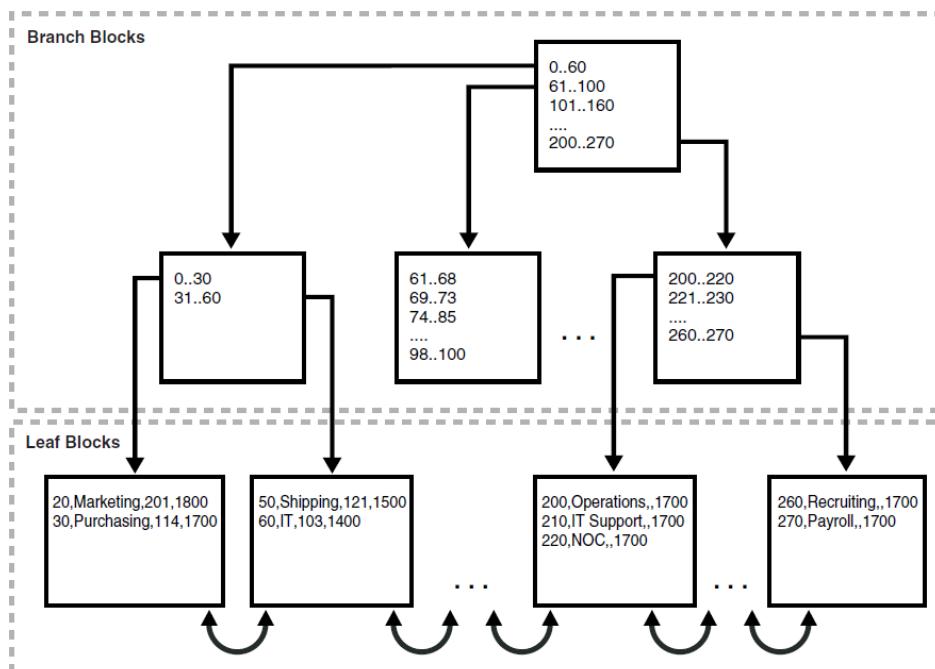
This part contains column values for the remaining non-key columns. This part is stored in the overflow storage area segment.

A secondary index is an index on an index-organized table. The secondary index is an independent schema object and is stored separately from the index-organized table. Rows in index leaf blocks can move within or between blocks because of insertions. Because rows in index-organized tables do not have permanent physical addresses, the database uses logical rowids based on primary key.

Secondary indexes use the logical rowids to locate table rows. A logical rowid includes a physical guess, which is the physical rowid of the index entry when it was first made.

With inaccurate physical guesses, access involves a secondary index scan and an I/O to fetch the wrong data block (as indicated by the guess), followed by an index unique scan of the index organized table by primary key value.

Figure 3-3 Index-Organized Table



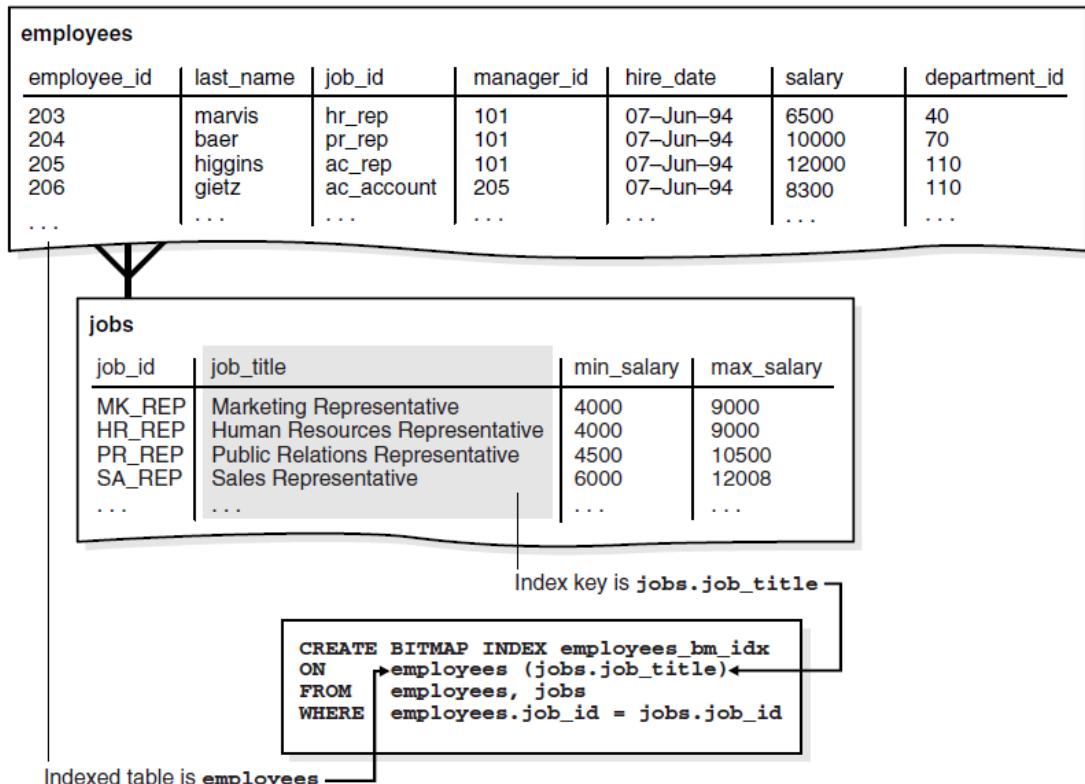
Bitmap Join Index

A bitmap join index is a bitmap index for the join of two or more tables. For each value in a table column, the index stores the rowid of the corresponding row in the indexed table.

An example when a bitmap join index would be useful:

```
SELECT COUNT(*) FROM employees, jobs
WHERE employees.job_id = jobs.job_id
AND jobs.job_title = 'Accountant';
```

Figure 3-2 Bitmap Join Index



jobs.job_title	employees.rowid
Accountant	AAAQNKAFAAAABSAAM
Accountant	AAAQNKAFAAAABSAAJ
Accountant	AAAQNKAFAAAABSAAK
Accounting Manager	AAAQNKAFAAAABTAAH
Administration Assistant	AAAQNKAFAAAABTAAC
Administration Vice President	AAAQNKAFAAAABSAAC
Administration Vice President	AAAQNKAFAAAABSAAB

```
CREATE BITMAP INDEX employees_bm_idx ON employees (jobs.job_title)
FROM employees, jobs WHERE employees.job_id=jobs.job_id;
```

You can find information about Bitmap Join indexes in the data dictionary:

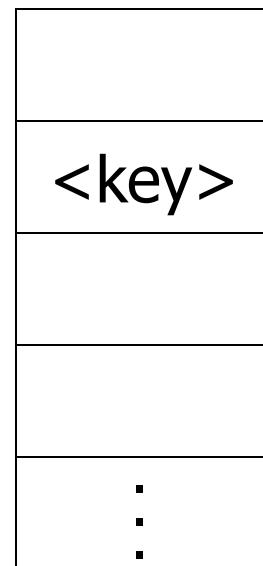
```
DBA_INDEXES.JOIN_INDEX → 'YES'
```

Ullman et al. : Database System Principles

Notes 5: Hashing and More

Hashing

$\text{key} \rightarrow h(\text{key})$



Buckets
(typically 1
disk block)

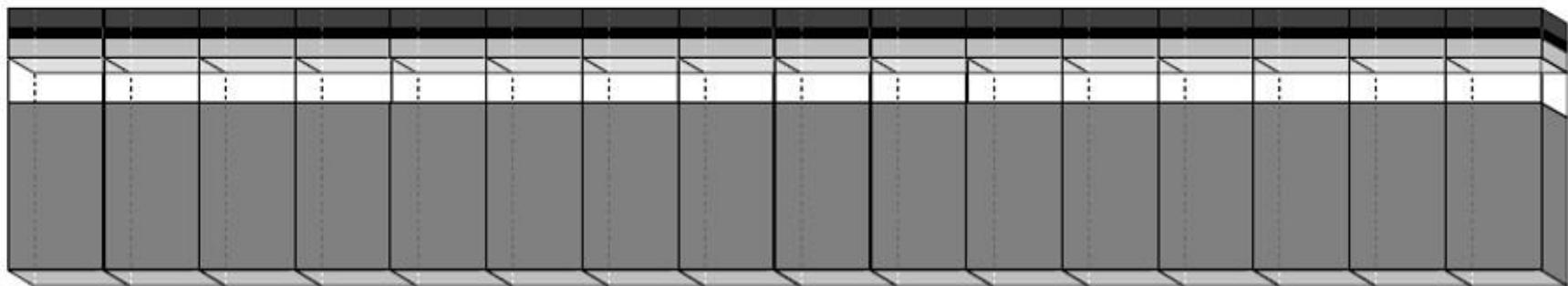
Hashing

```
SELECT * FROM employees  
WHERE department_id = 20
```

Hash(20) → Hash Value 77

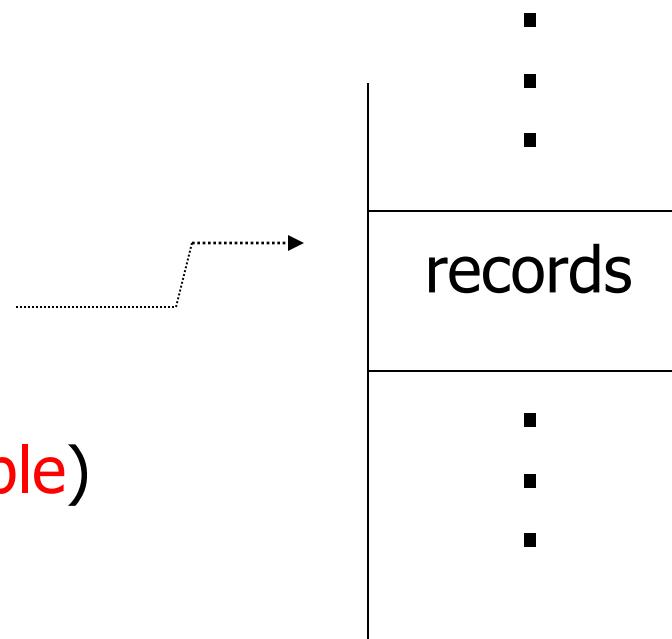
Block
100

Data Blocks



Two alternatives

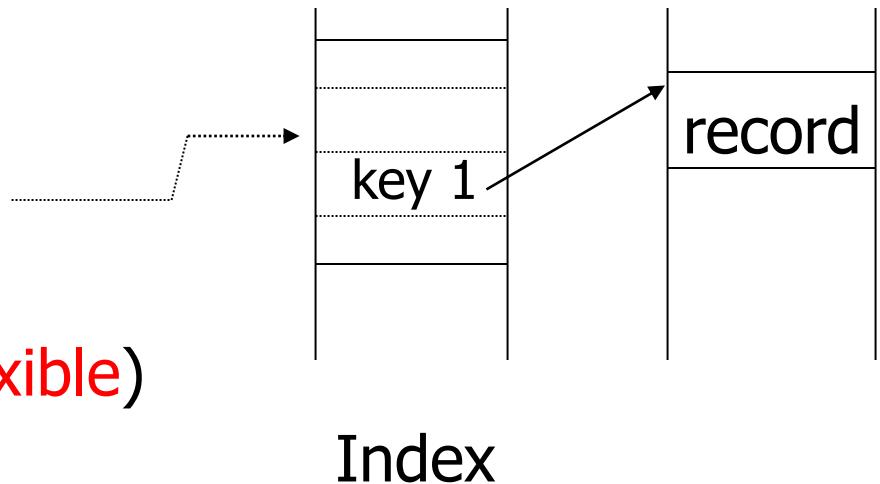
(1) $\text{key} \rightarrow h(\text{key})$
(direct reference, **not flexible**)



Two alternatives

(2) $\text{key} \rightarrow h(\text{key})$

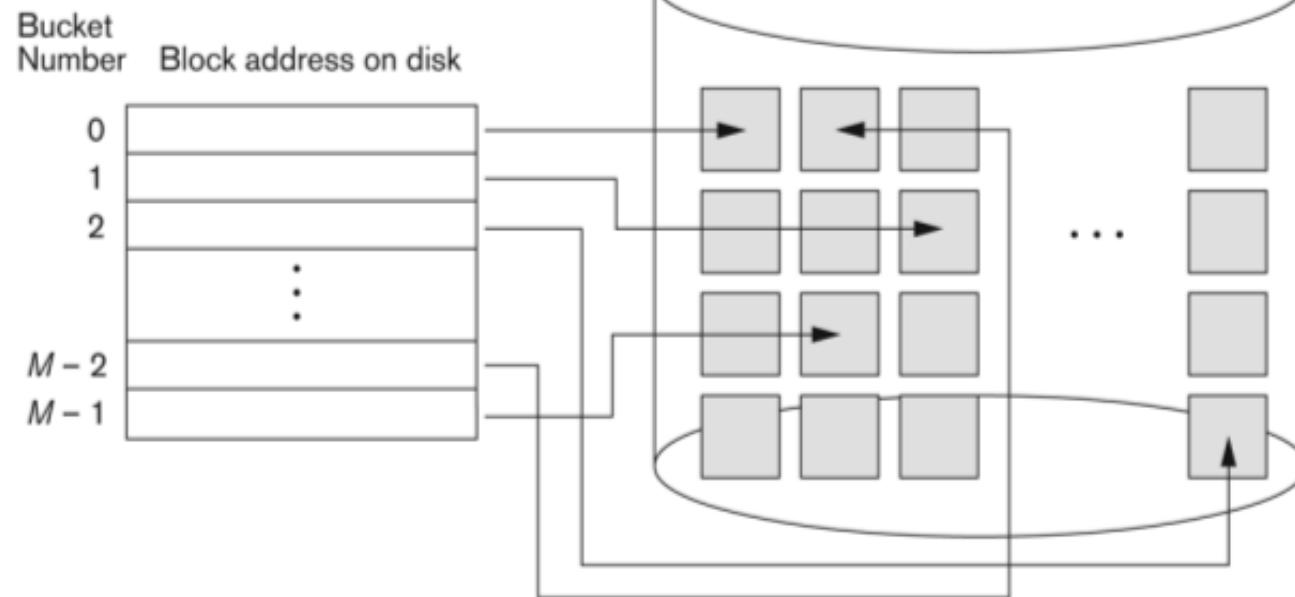
(indirect reference, **more flexible**)



- Alt (2) for “secondary” search key

Typical implementation

Matching bucket numbers to disk block addresses.



Example hash function

- Key = ' $x_1 x_2 \dots x_n$ ' n byte character string
- Have b buckets
- h : add $x_1 + x_2 + \dots + x_n$
 - compute sum modulo b

- ☒ This may not be best function ...
- ☒ Read Knuth Vol. 3 if you really need to select a good function.

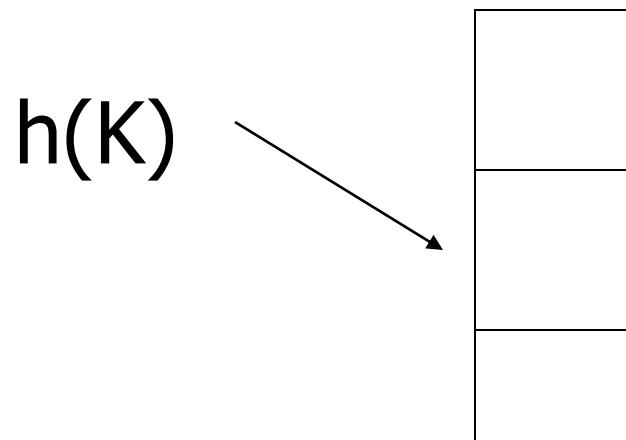
Good hash
function:

☞ Expected number of
keys/bucket is the
same for all buckets

Within a bucket:

- Do we keep keys sorted?
- Yes, if CPU time critical
& Inserts/Deletes not too frequent

Next: example to illustrate
inserts, overflows, deletes



EXAMPLE 2 records/bucket

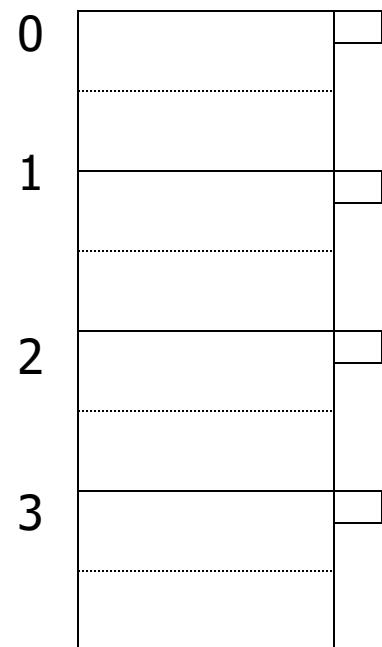
INSERT:

$$h(a) = 1$$

$$h(b) = 2$$

$$h(c) = 1$$

$$h(d) = 0$$



EXAMPLE 2 records/bucket

INSERT:

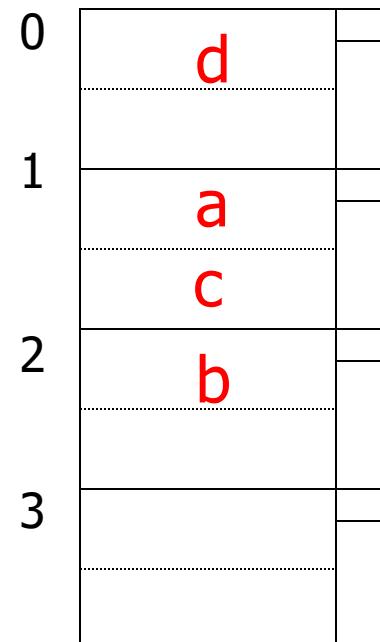
$$h(a) = 1$$

$$h(b) = 2$$

$$h(c) = 1$$

$$h(d) = 0$$

$$h(e) = 1$$



EXAMPLE 2 records/bucket

INSERT:

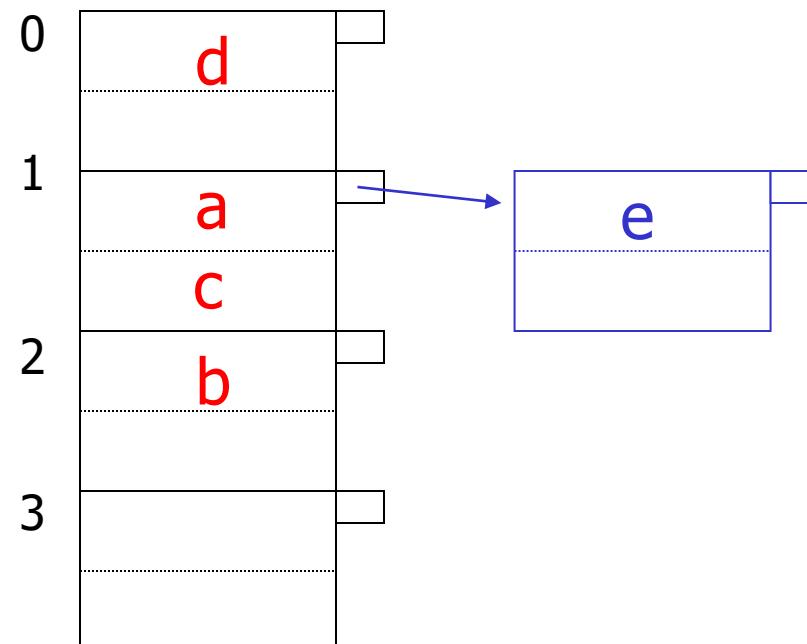
$$h(a) = 1$$

$$h(b) = 2$$

$$h(c) = 1$$

$$h(d) = 0$$

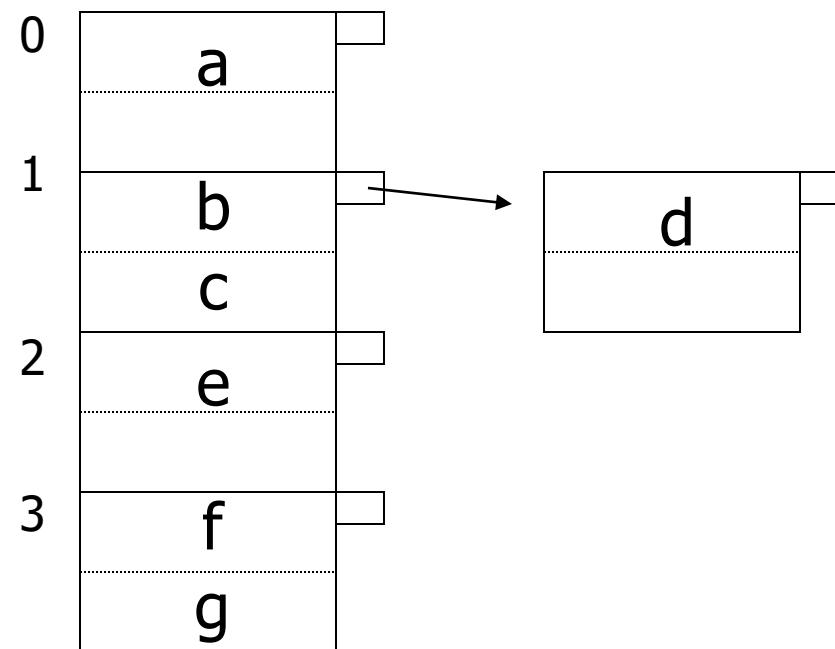
$$h(e) = 1$$



EXAMPLE: deletion

Delete:

e
f



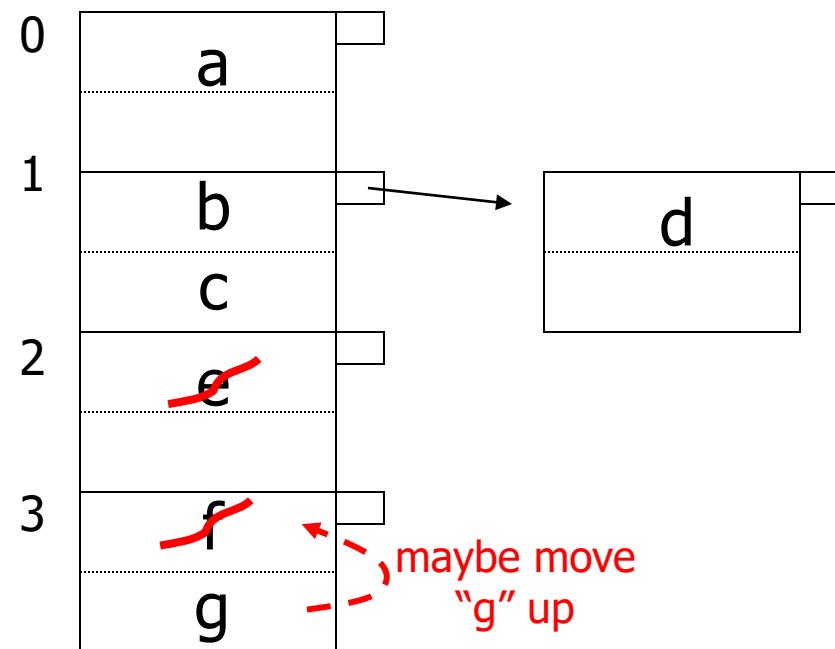
EXAMPLE: deletion

Delete:

e

f

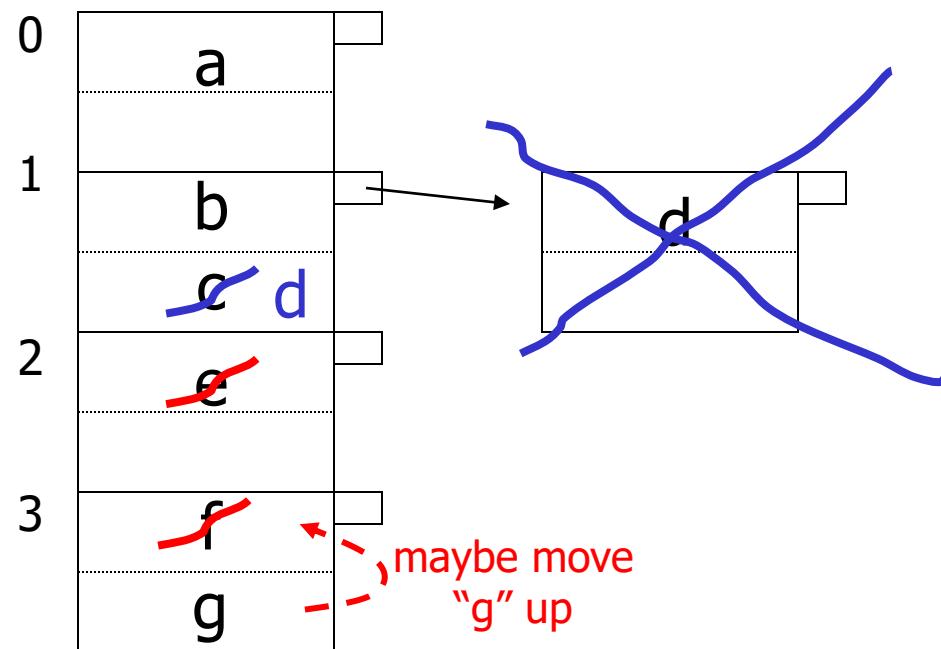
C



EXAMPLE: deletion

Delete:

e
f
C



Rule of thumb:

- Try to keep space utilization between 50% and 80%

$$\text{Utilization} = \frac{\text{\# keys used}}{\text{total \# keys that fit}}$$

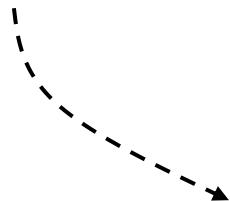
- If < 50%, wasting space
- If > 80%, overflows significant
 - depends on how good hash function is & on # keys/bucket

How do we cope with growth?

- 
- Overflows and reorganizations
 - Dynamic hashing

How do we cope with growth?

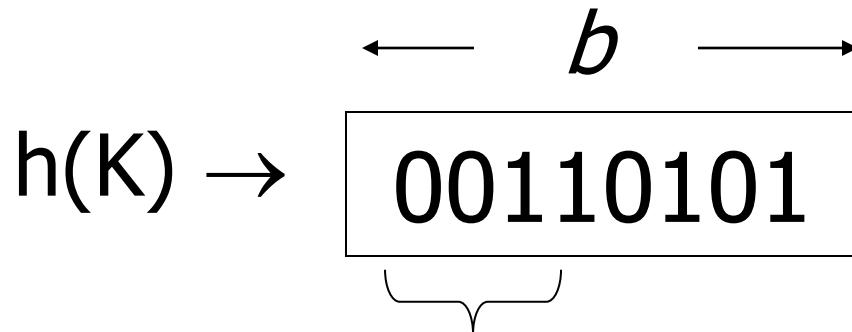
- Overflows and reorganizations
- **Dynamic hashing**



- Extensible
- Linear

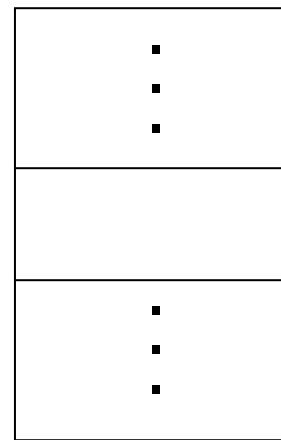
Extensible hashing: two ideas

(a) Use i of b bits output by hash function



(b) Use directory

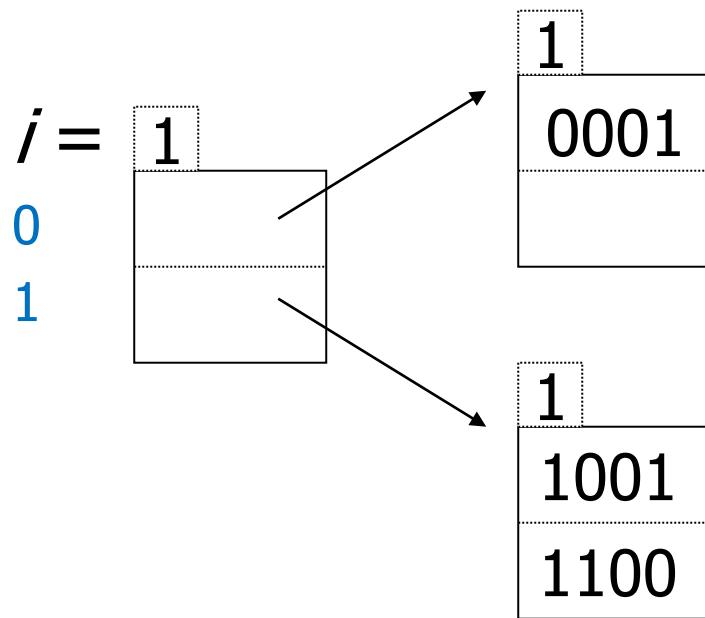
$h(K)[i]$



to bucket

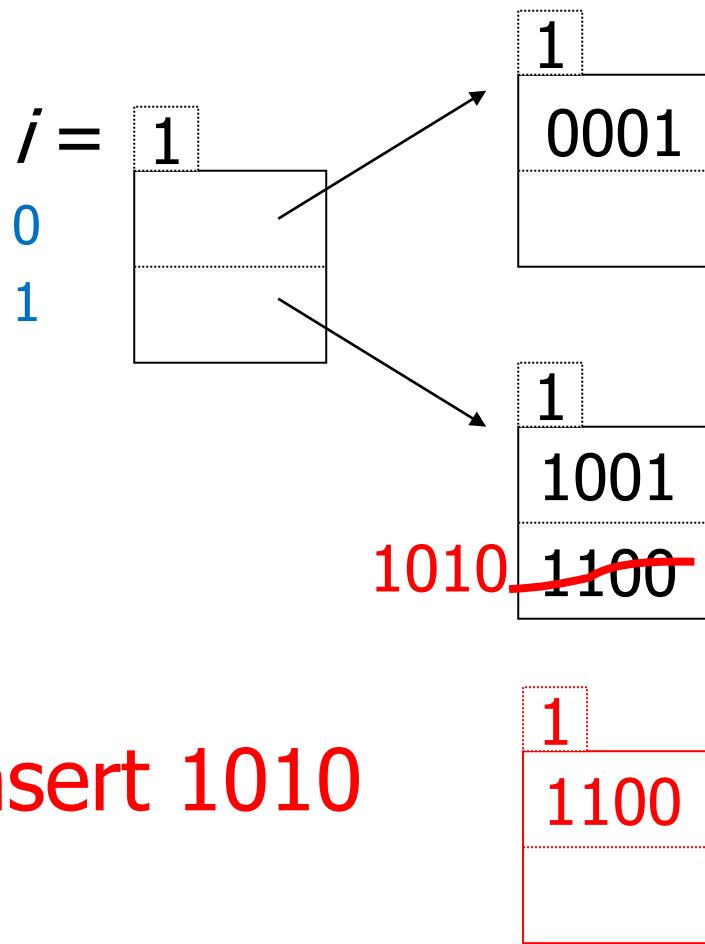
$h(K)[i]$: means the first i bits of the output by hash function

Example: $h(k)$ is 4 bits; 2 keys/bucket

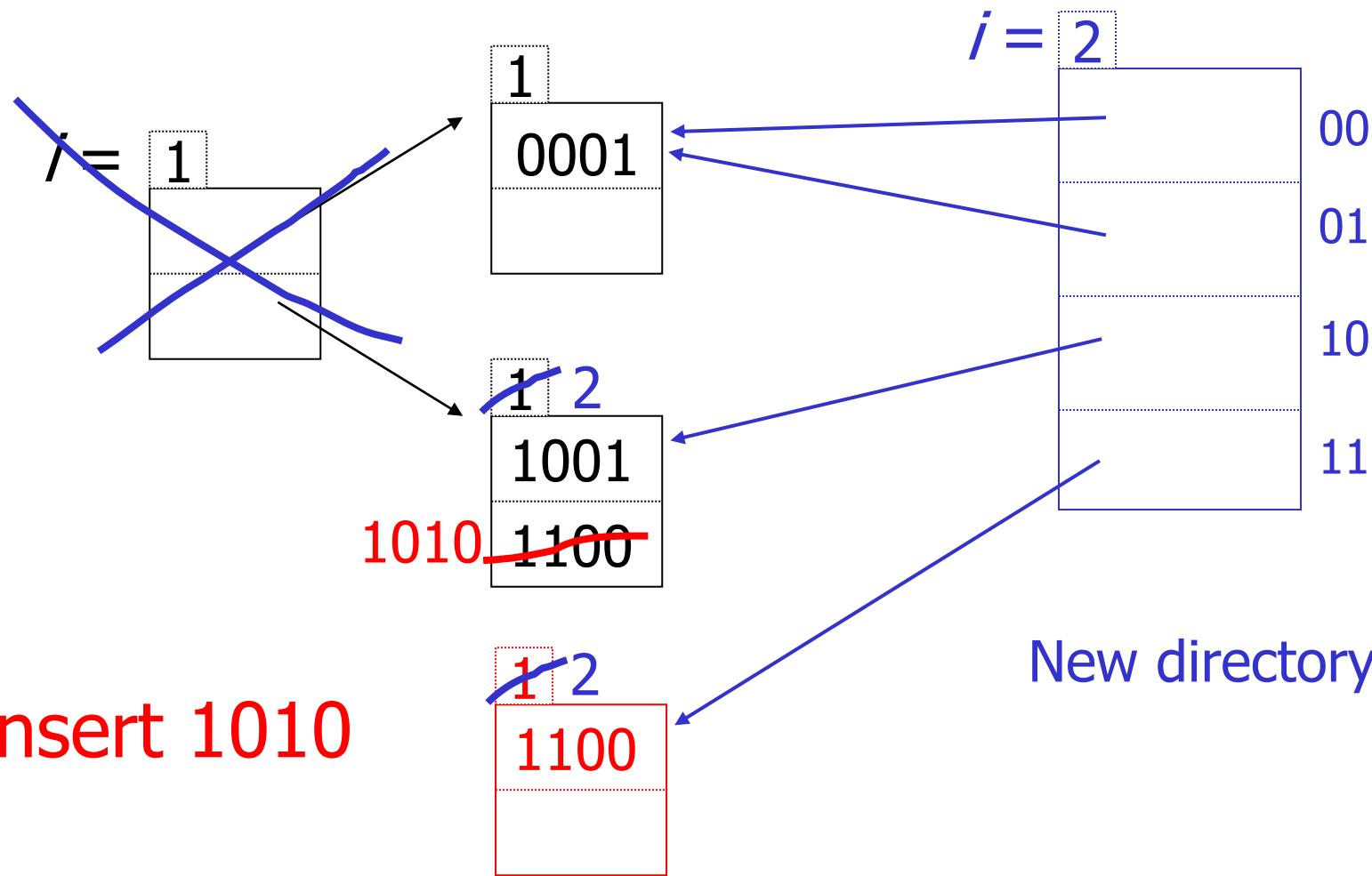


Insert 1010

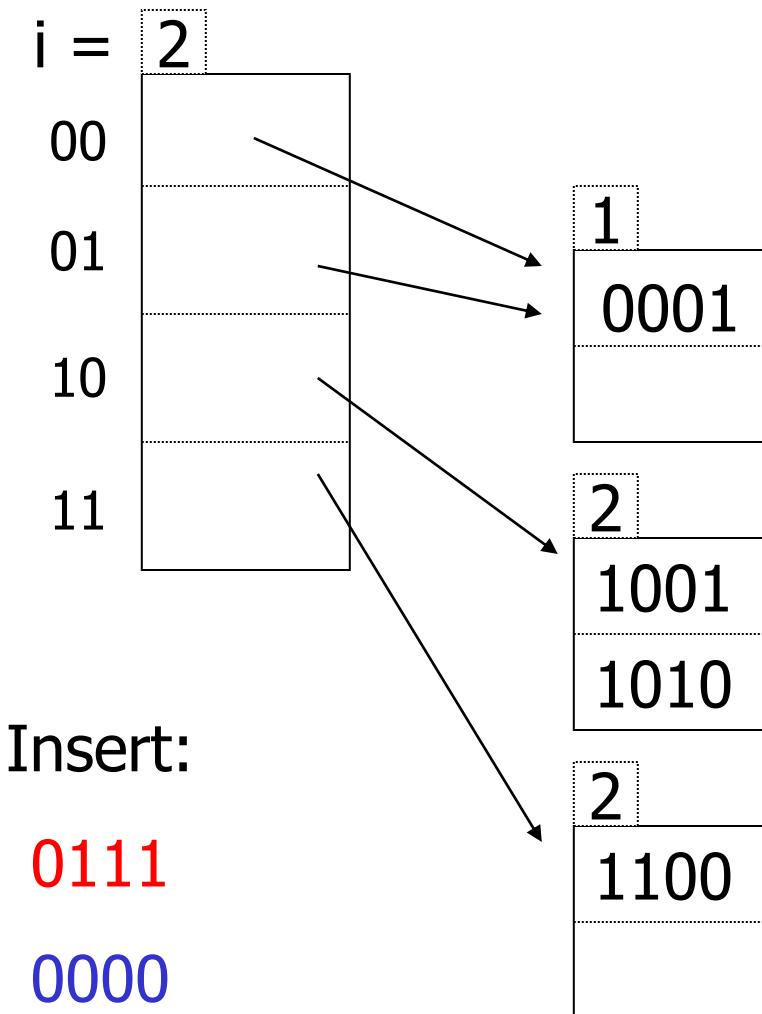
Example: $h(k)$ is 4 bits; 2 keys/bucket



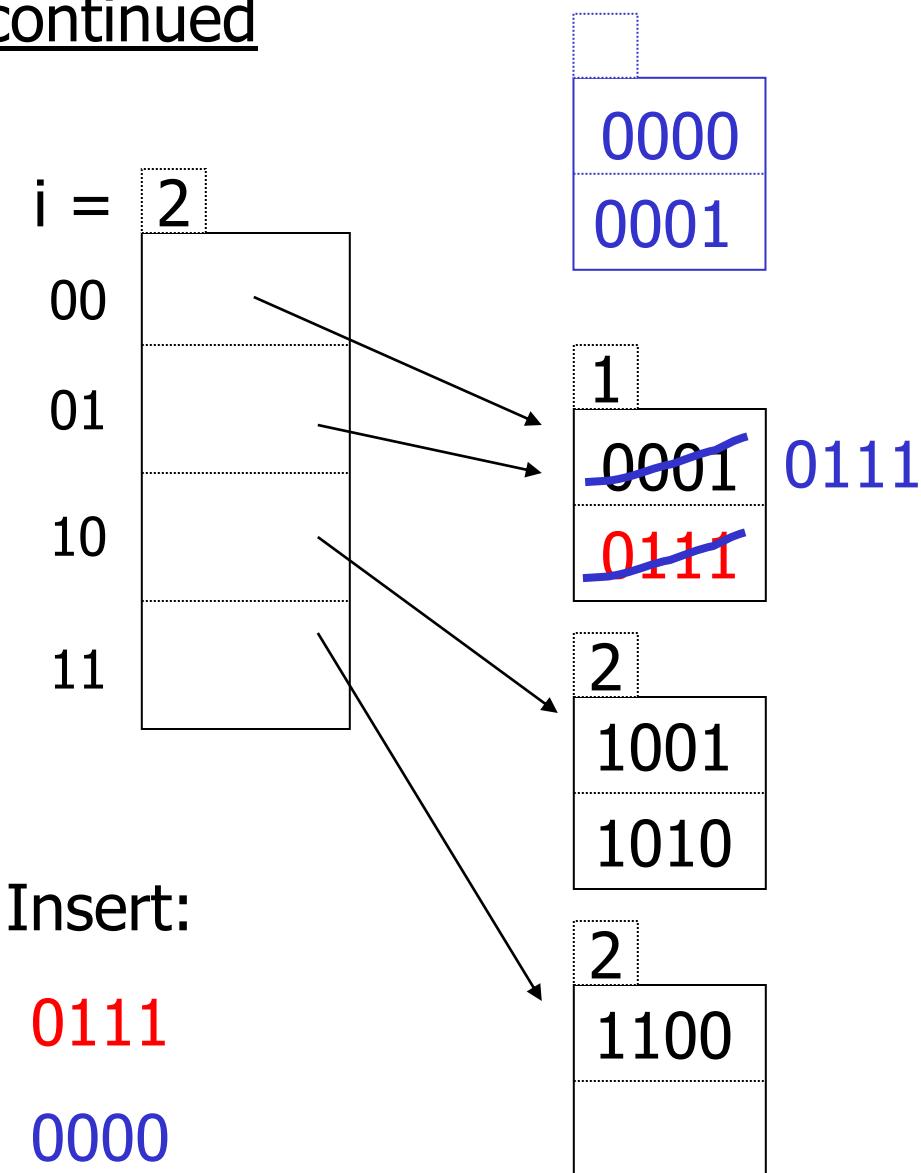
Example: $h(k)$ is 4 bits; 2 keys/bucket



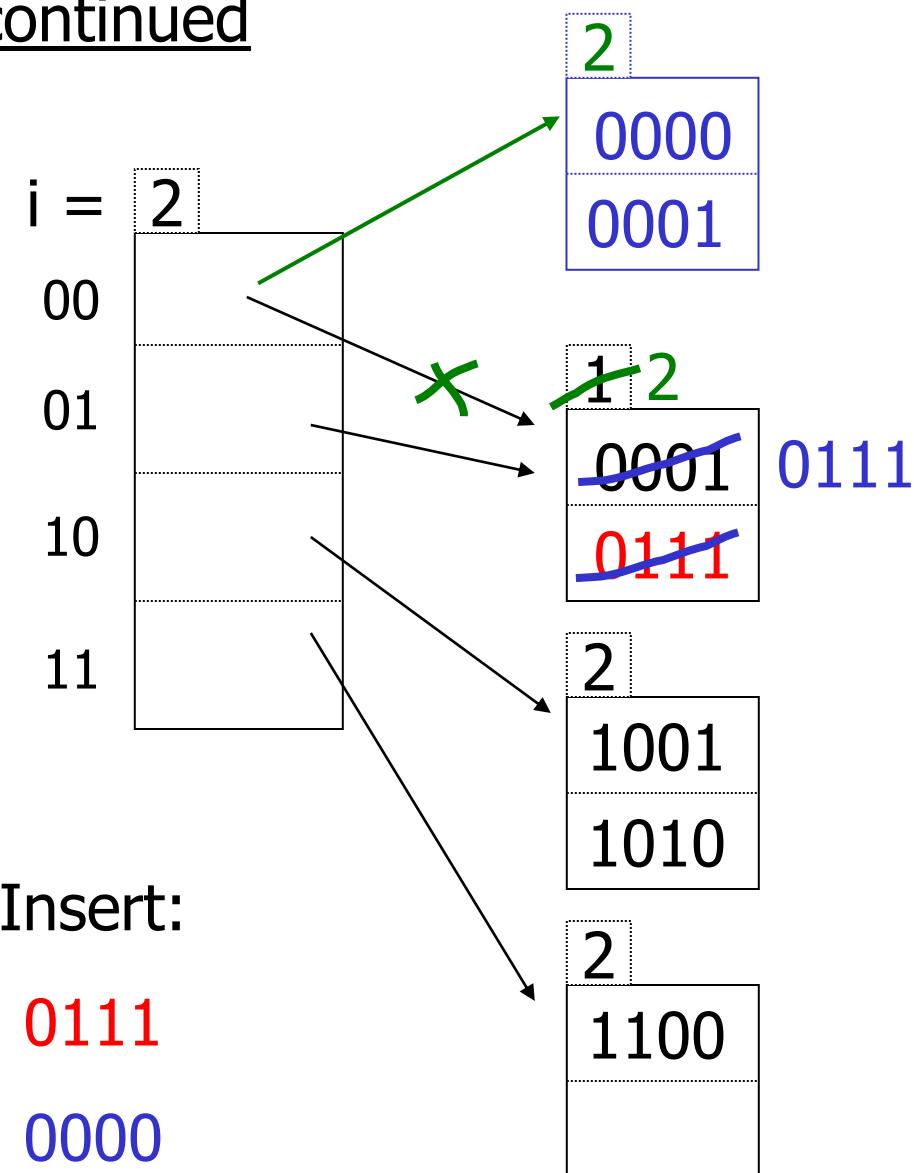
Example continued



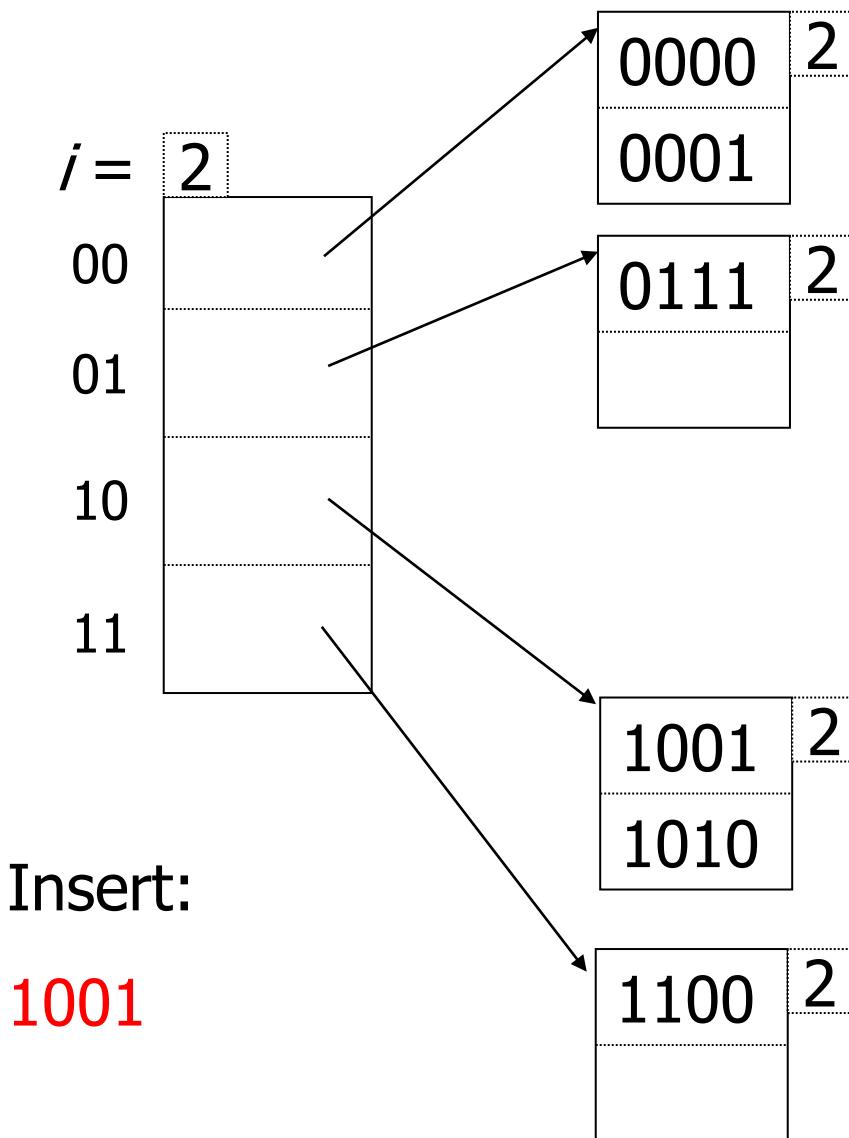
Example continued



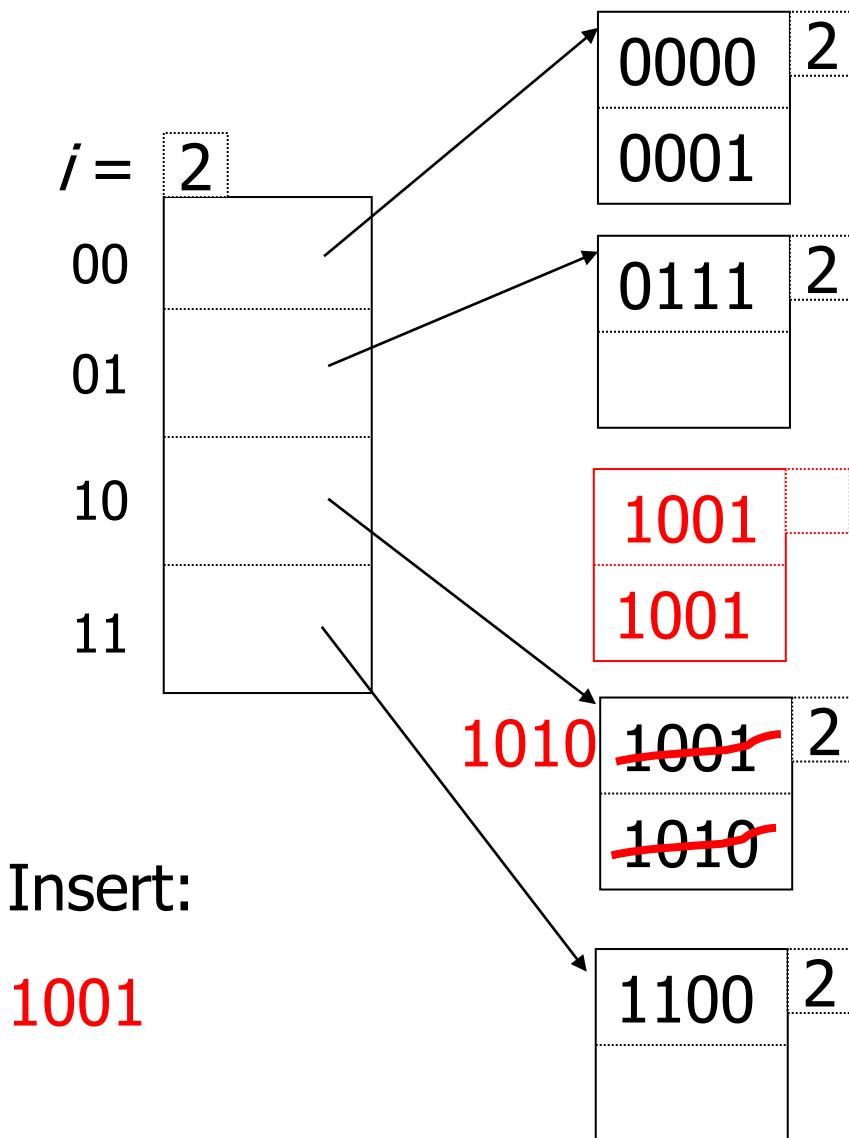
Example continued



Example continued



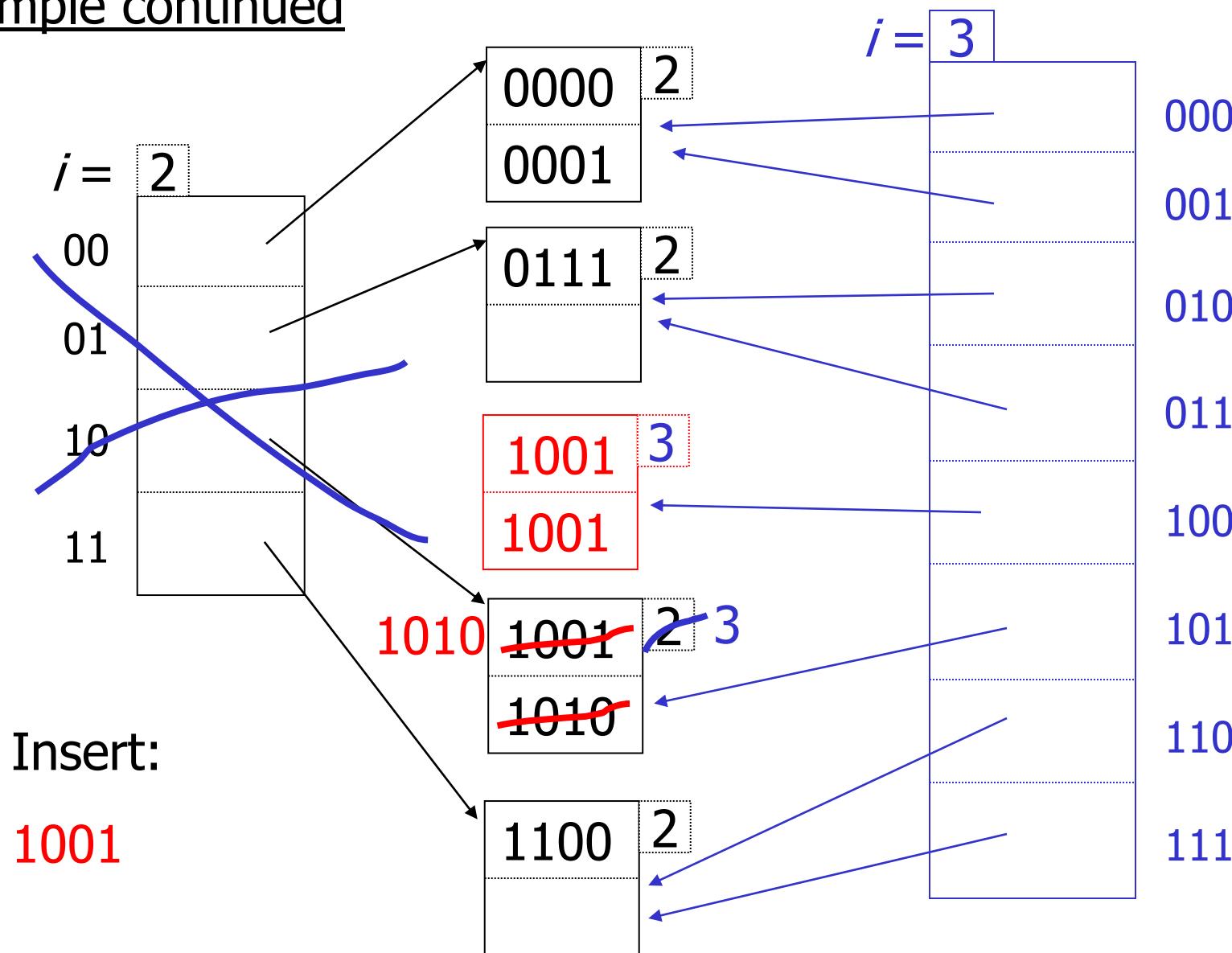
Example continued



Insert:

1001

Example continued



Extensible hashing: deletion

- No merging of blocks
- Merge blocks
and cut directory if possible
(Reverse insert procedure)

Deletion example:

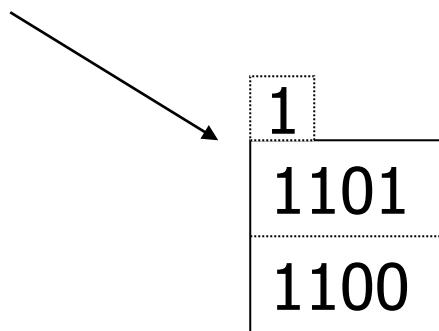
- Run thru insert example in reverse!

But: Typically **not implemented**

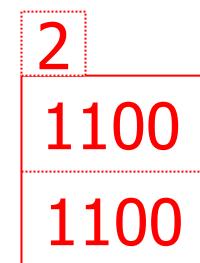
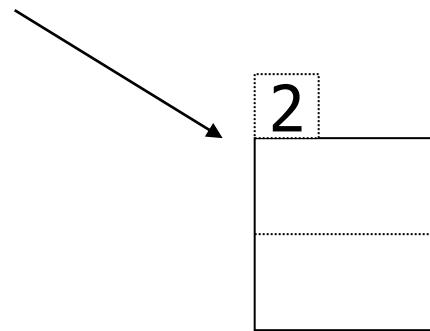
Note: Still need overflow chains

- Example: many records with duplicate keys

insert 1100

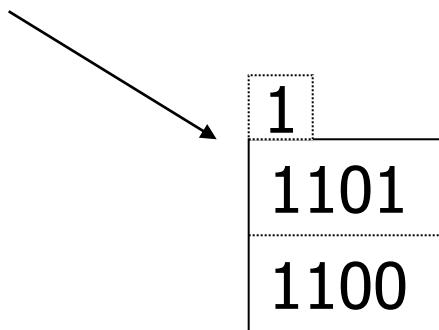


if we split:

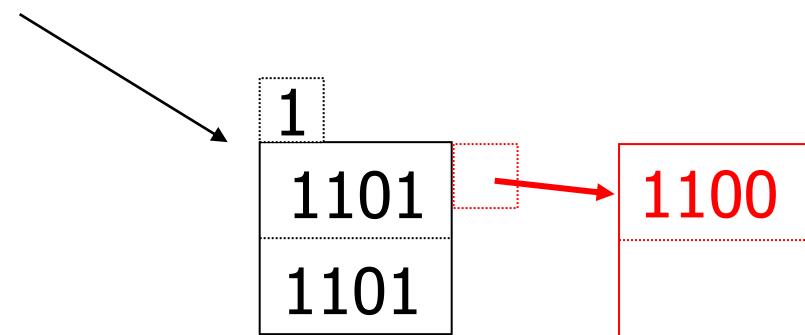


Solution: overflow chains

insert 1100



add overflow block:



Summary

Extensible hashing

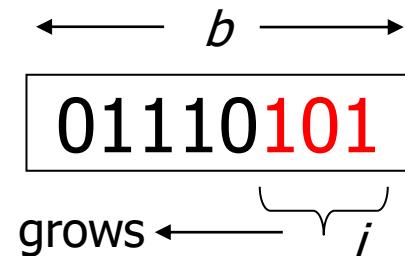
- ⊕ Can handle growing files
 - with less wasted space
 - with no full reorganizations
- ⊖ Indirection
 - (Not bad if directory in memory)
- ⊖ Directory doubles in size
 - (Now it fits, now it does not)

Linear hashing

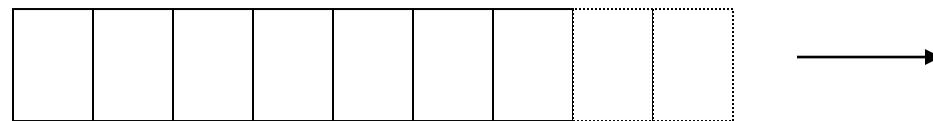
- Another dynamic hashing scheme

Two ideas:

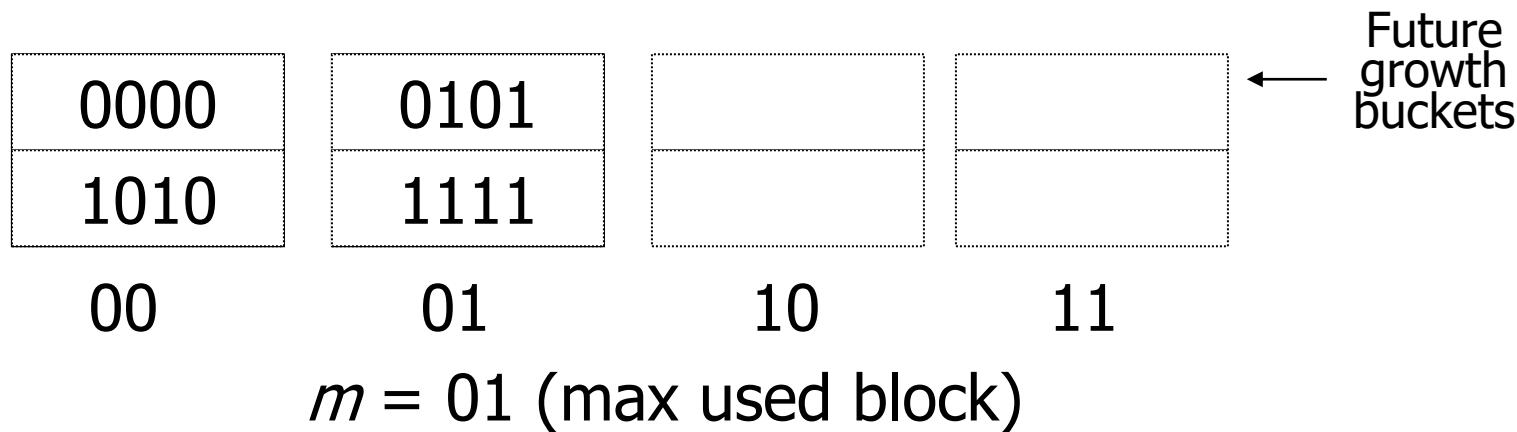
(a) Use i low order bits of hash



(b) File grows linearly



Example $b=4$ bits, $i=2$, 2 keys/bucket



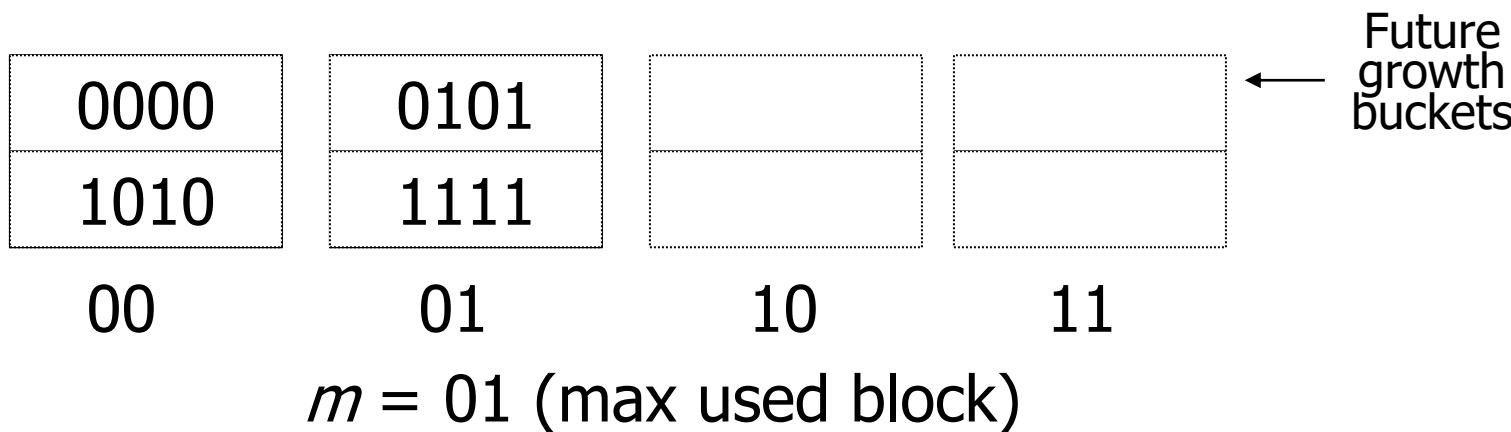
Rule If $h(k)[i] \leq m$, then

look at bucket $h(k)[i]$

else, look at bucket $h(k)[i] - 2^{i-1}$

Example $b=4$ bits, $i=2$, 2 keys/bucket

- insert 0101

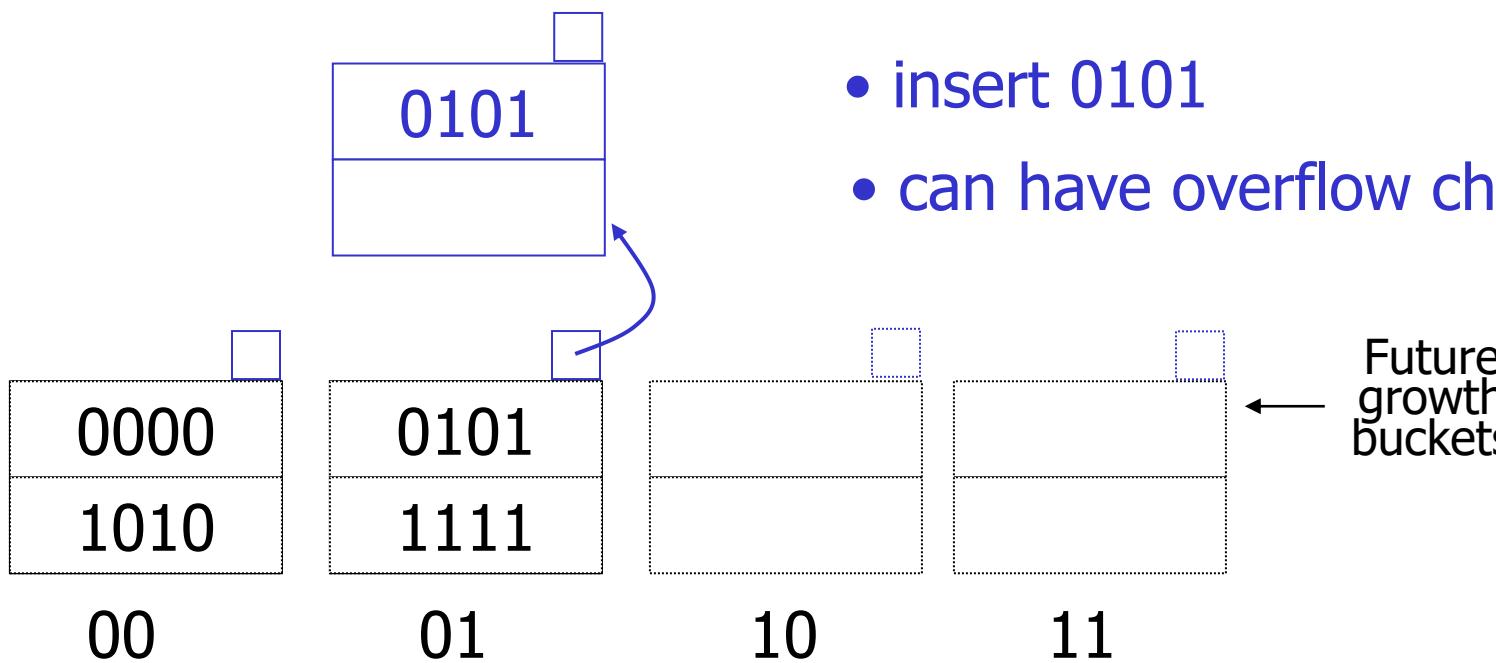


Rule If $h(k)[i] \leq m$, then

look at bucket $h(k)[i]$

else, look at bucket $h(k)[i] - 2^{i-1}$

Example $b=4$ bits, $i=2$, 2 keys/bucket

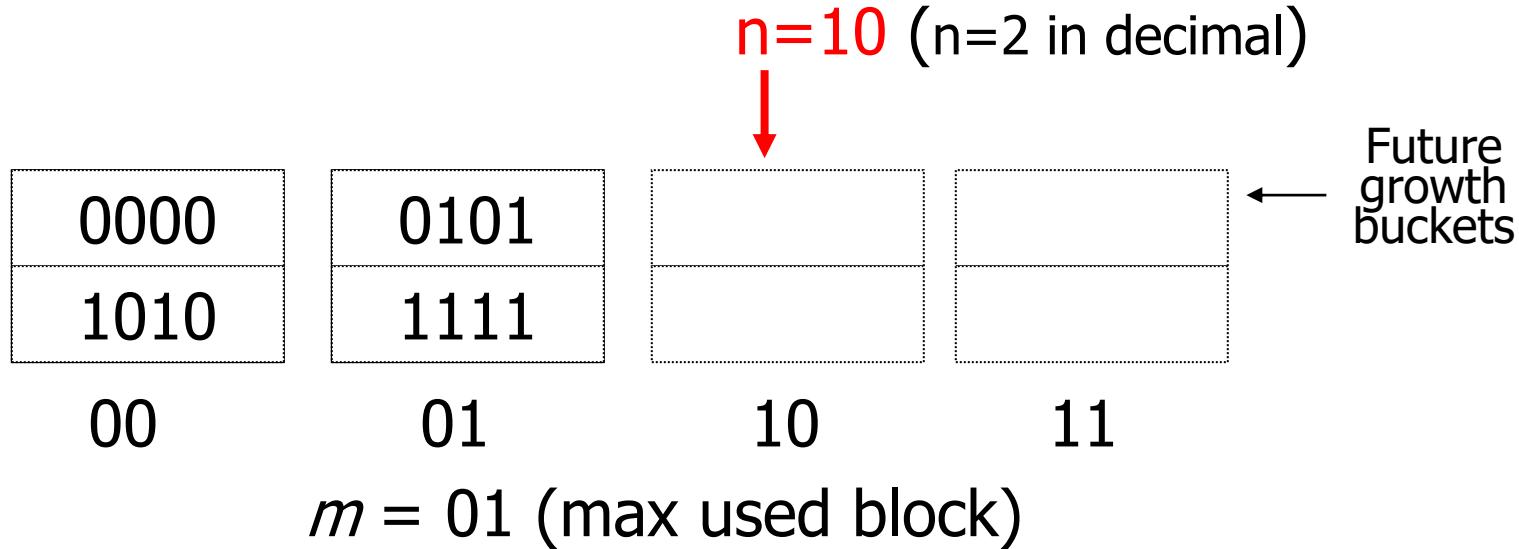


Rule If $h(k)[i] \leq m$, then

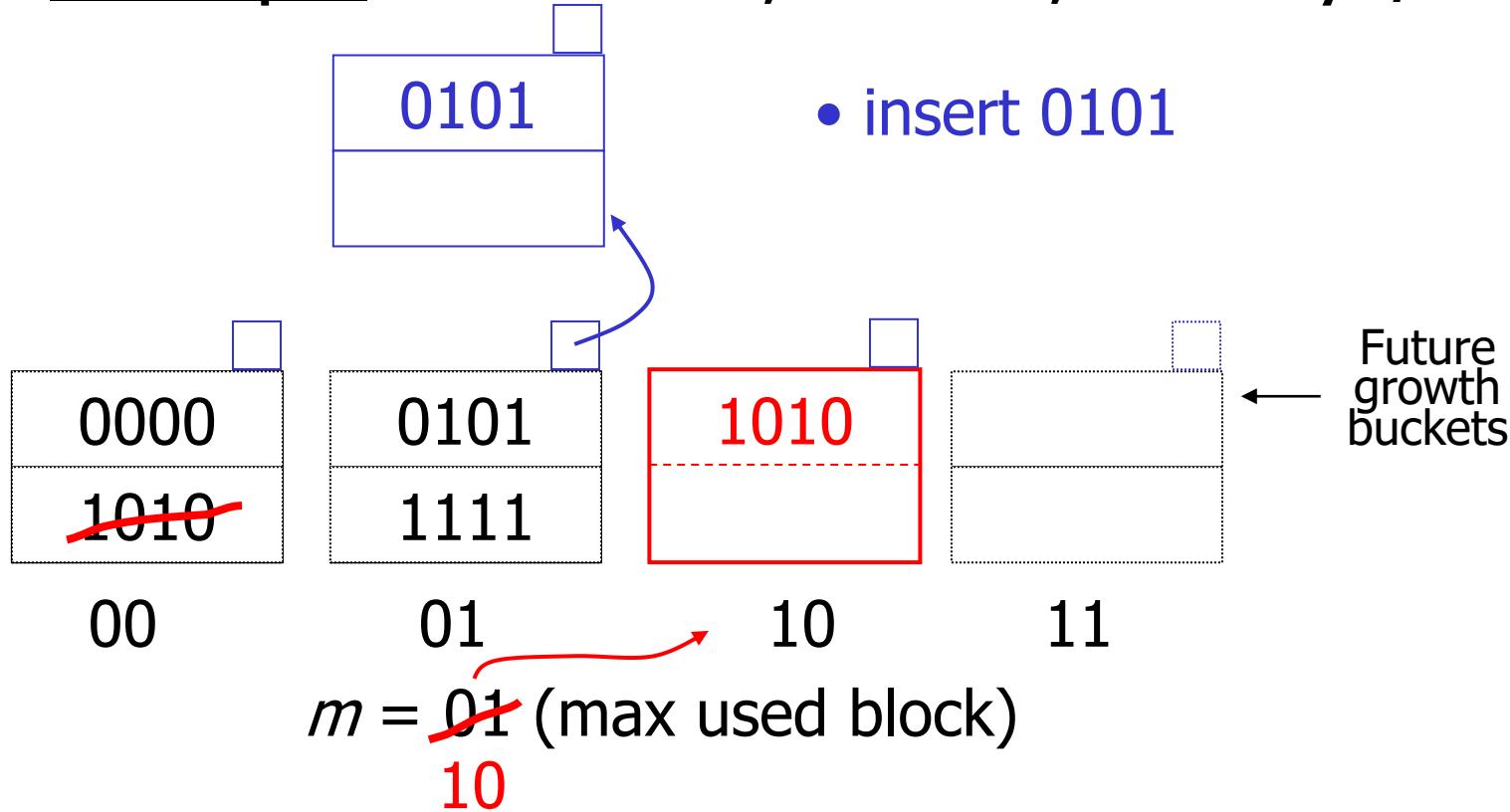
look at bucket $h(k)[i]$
else, look at bucket $h(k)[i] - 2^{i-1}$

Note

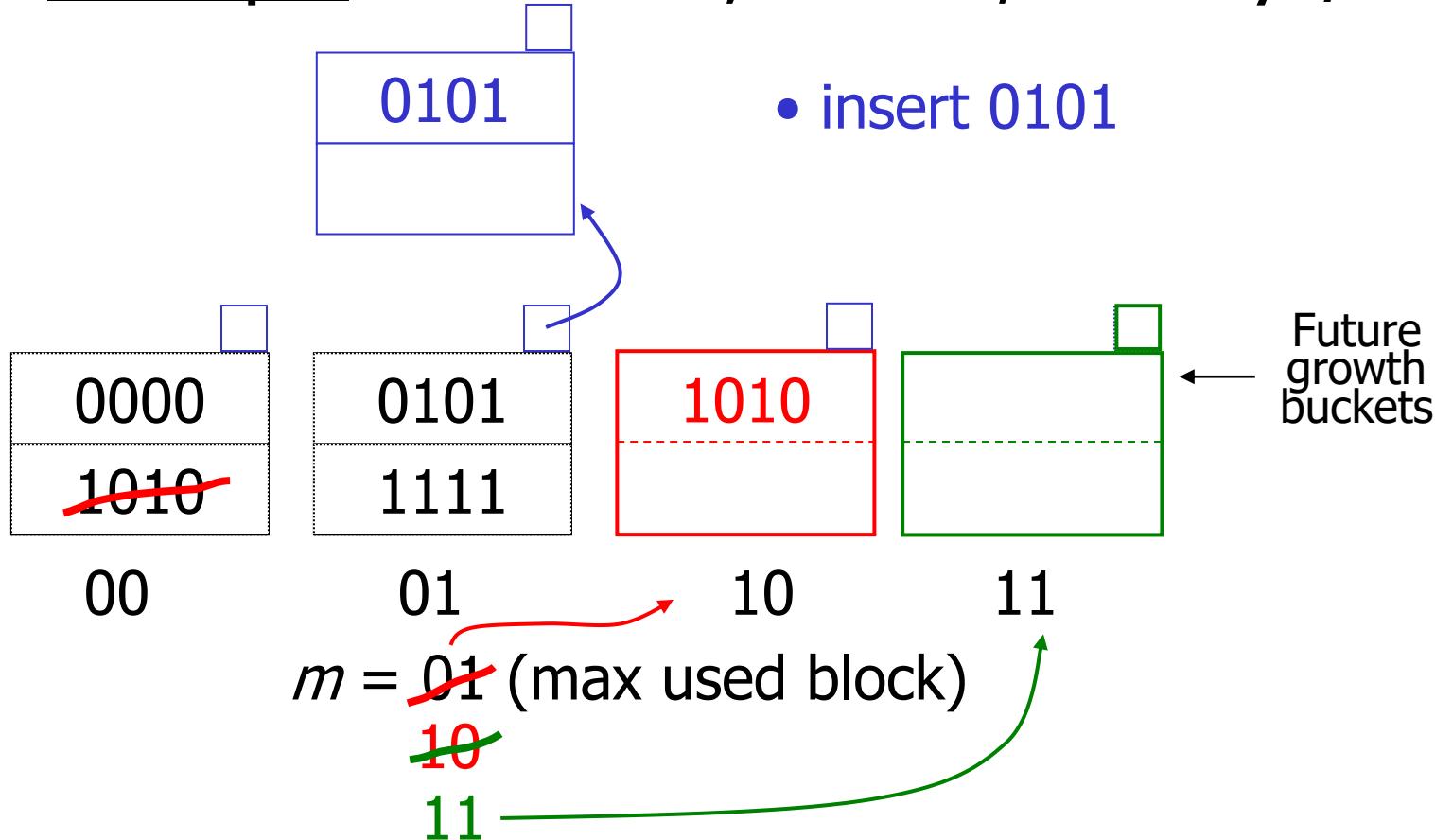
- In textbook, n is used instead of m
- $n=m+1$ (n =number of used blocks)



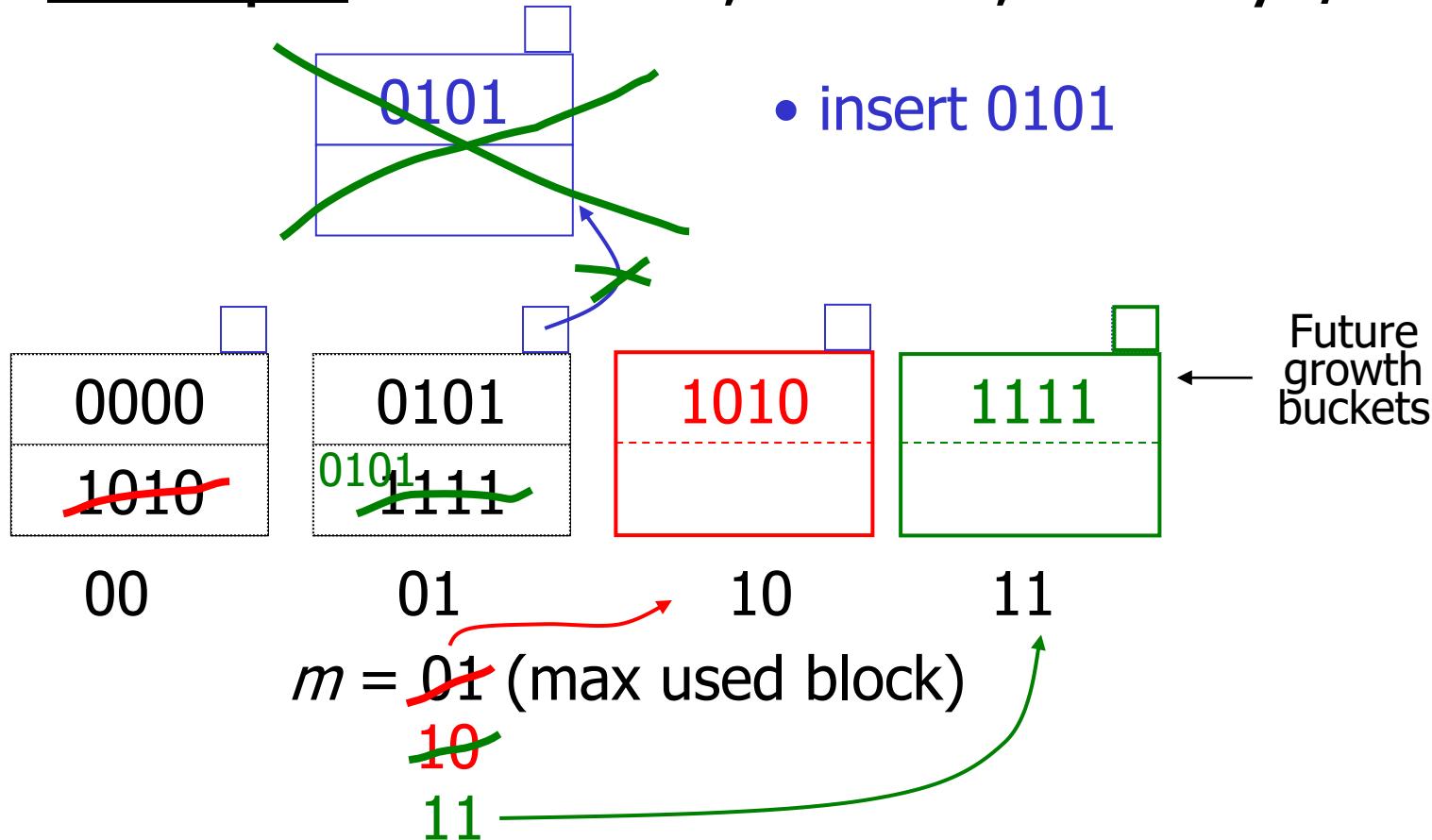
Example $b=4$ bits, $i=2$, 2 keys/bucket



Example $b=4$ bits, $i=2$, 2 keys/bucket

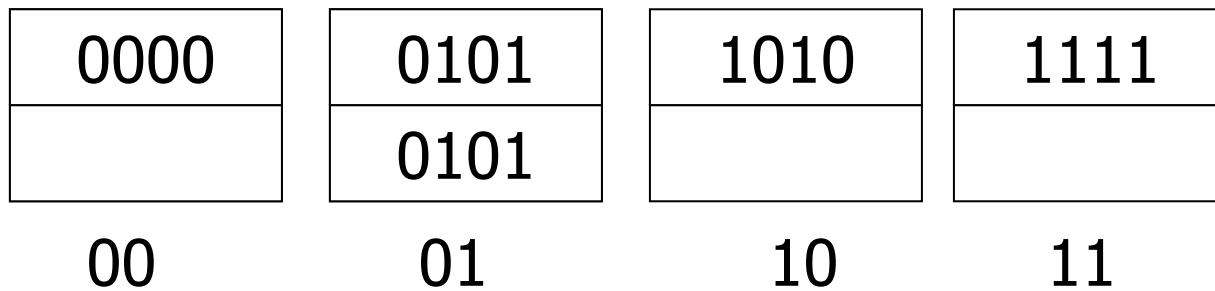


Example $b=4$ bits, $i=2$, 2 keys/bucket



Example Continued: How to grow beyond this?

$i = 2$



$m = 11$ (max used block)

Example Continued: How to grow beyond this?

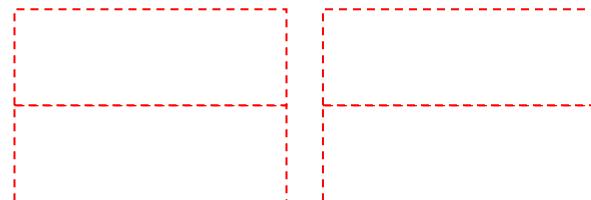
$i = \cancel{2} 3$

0000

0101
0101

1010

1111



000
100

001
101

010
110

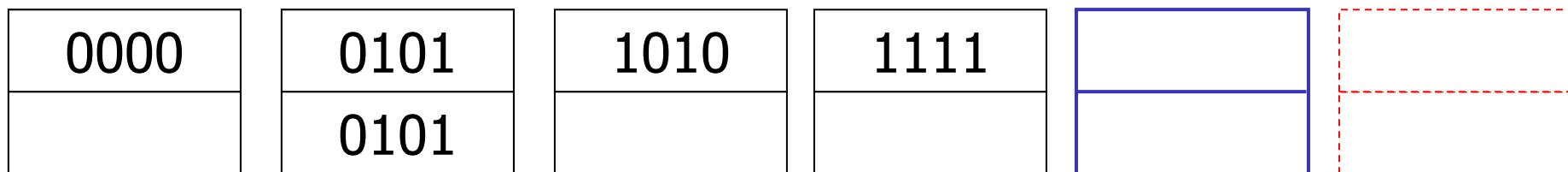
011
111

...

$m = 11$ (max used block)

Example Continued: How to grow beyond this?

$i = \cancel{2} 3$



~~000
100~~

001
101

010
110

011
111

100

...

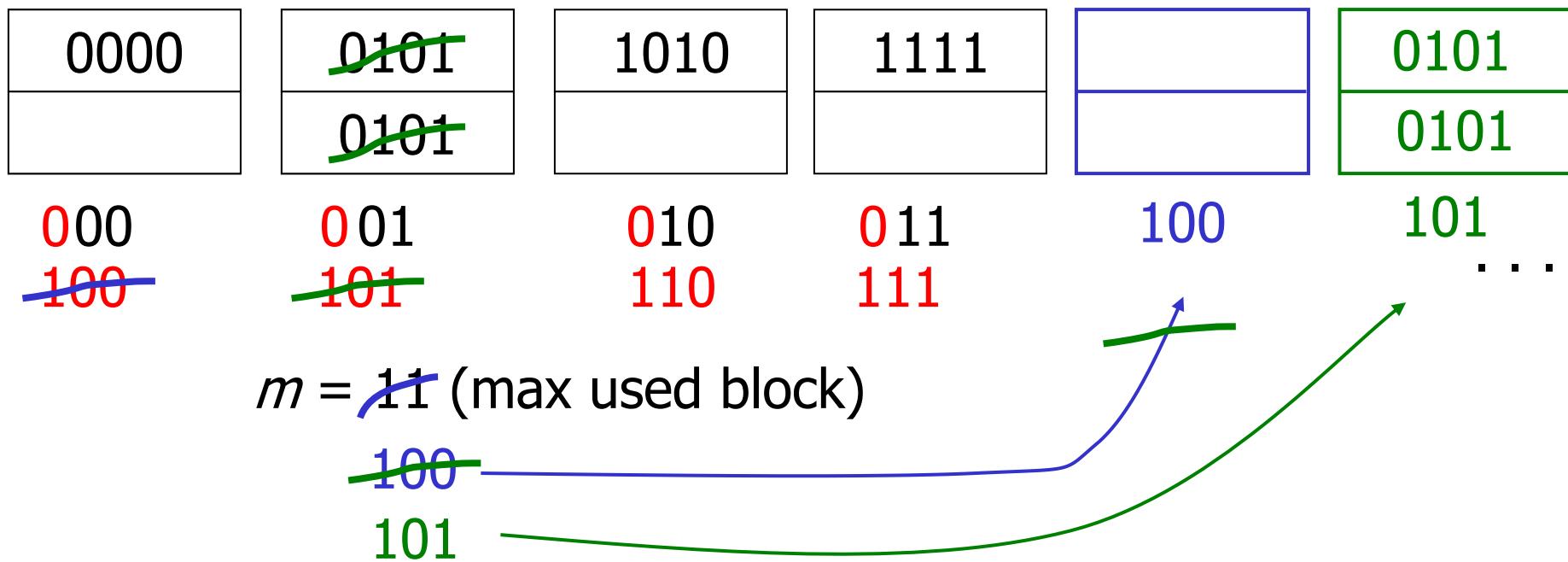
$m = \cancel{11}$ (max used block)

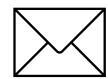
100



Example Continued: How to grow beyond this?

$i = \cancel{2} 3$





When do we expand file?

- Keep track of:
$$\frac{\text{# used slots}}{\text{total # of slots}} = U$$

#used slots → #records, total # of slots → #buckets

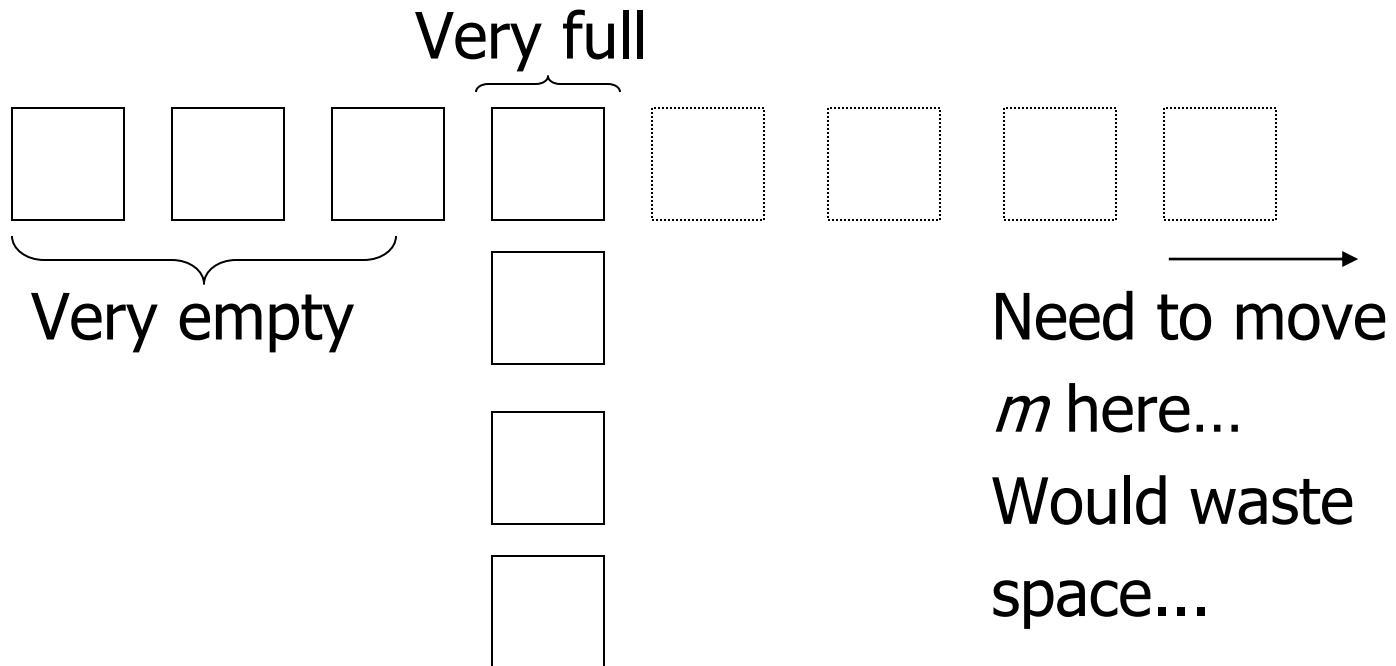
- If $U >$ **threshold** then increase m
(and maybe i)

Summary

Linear Hashing

- ⊕ Can handle growing files
 - with **less wasted space**
 - with no full reorganizations
- ⊕ **No indirection** like extensible hashing
- ⊖ Can still have **overflow chains**

Example: **BAD CASE**



Hashing depends on data distribution!

Summary

Hashing

- How it works
- Dynamic hashing
 - Extensible
 - Linear

Next:

- Indexing vs Hashing
- Index definition in SQL
- Multiple key access

Indexing vs Hashing

- Hashing good for probes **given key**

e.g., **SELECT ...**

FROM R

WHERE R.A = 5

Indexing vs Hashing

- INDEXING (Including B Trees) good for Range Searches:

e.g.,

SELECT FROM R

WHERE R.A > 5 AND R.A < 10;

Index definition **in SQL**

- Create index name on rel (attr)
- Create unique index name on rel (attr)
 - defines candidate key
- Drop INDEX name

Note

CANNOT SPECIFY TYPE OF INDEX

(e.g. B-tree, Hashing, ...)

OR PARAMETERS

(e.g. Load Factor, Size of Hash,...)

... at least in SQL...

In Oracle you can !

Note

ATTRIBUTE LIST \Rightarrow MULTIKEY INDEX
(next)

e.g., CREATE INDEX foo ON R(A,B,C)

Multi-key Index

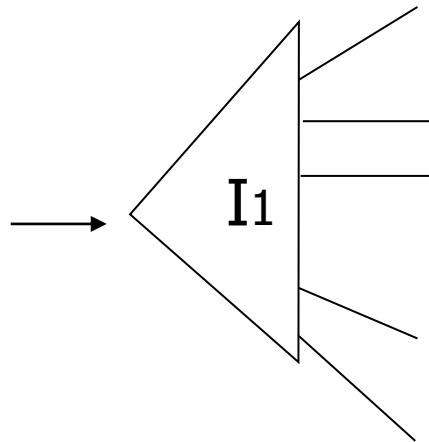
Motivation: Find records where

DEPT = "Toy" AND SAL > 50k

What kind of indexes can support this query?

Strategy I:

- Use one index, say Dept.
- Get all Dept = "Toy" records and check their salary



Strategy II:

- Use 2 Indexes; Manipulate Pointers

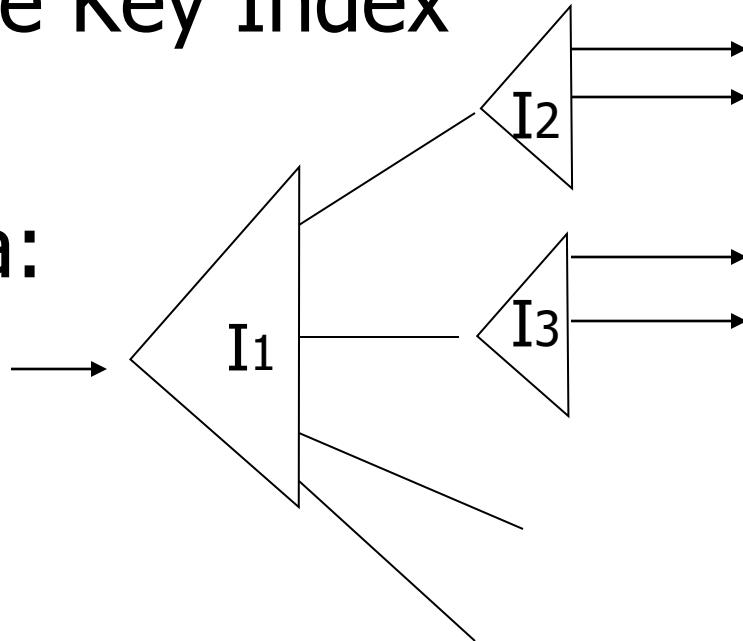
Toy →  ← Sal > 50k

AND → intersection of pointers

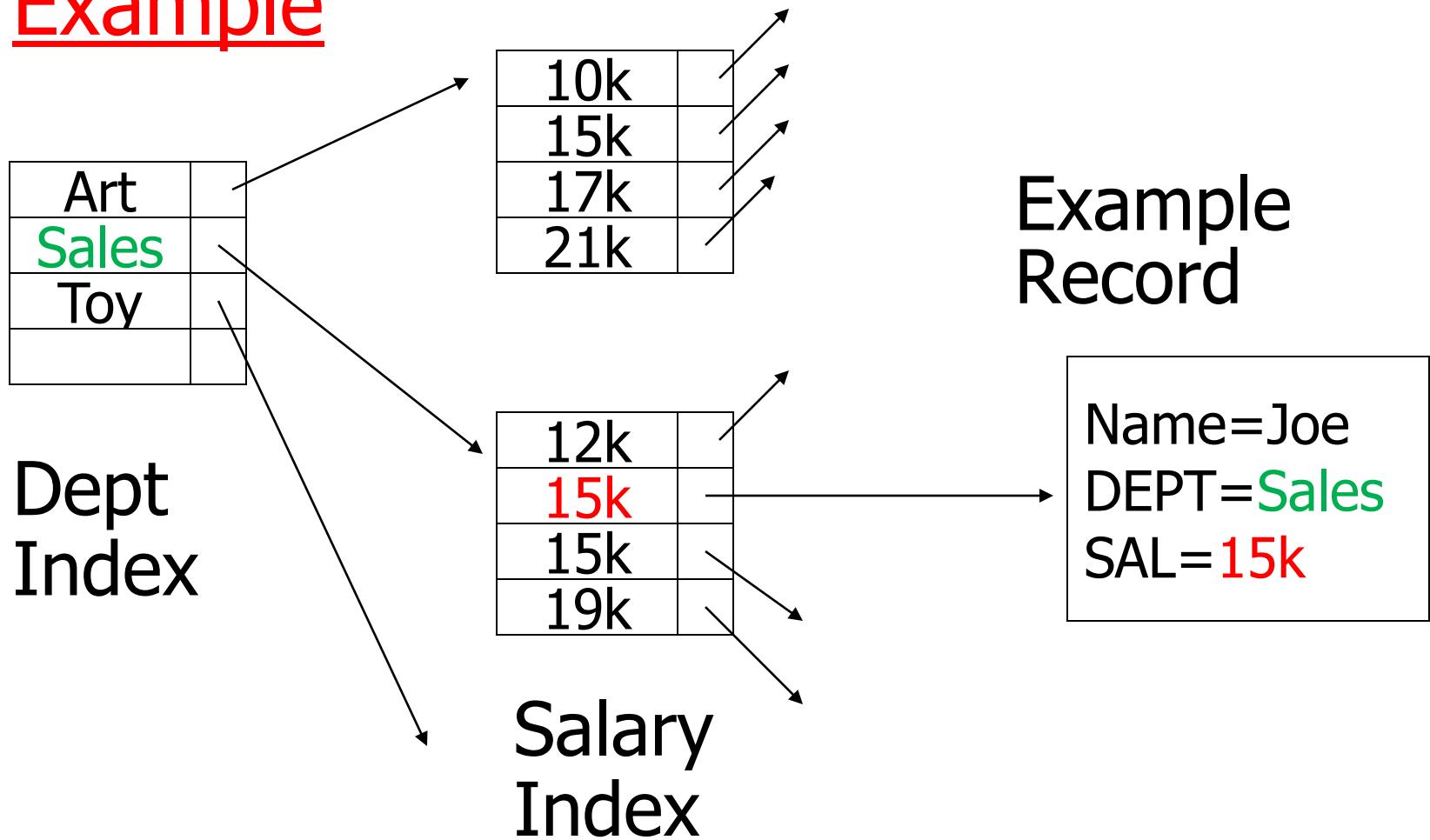
Strategy III:

- Multiple Key Index

One idea:



Example



For which queries is this index good?

- Find RECs Dept = "Sales" \wedge SAL=20k
- Find RECs Dept = "Sales" \wedge SAL \geq 20k
- Find RECs Dept = "Sales"
- Find RECs SAL = 20k

Query processing and optimization

Definitions

- **Query processing**
 - translation of query into low-level activities
 - evaluation of query
 - data extraction (read data from file and implement operations)
- **Query optimization**
 - selecting the most efficient query evaluation

Query Processing (1/2)

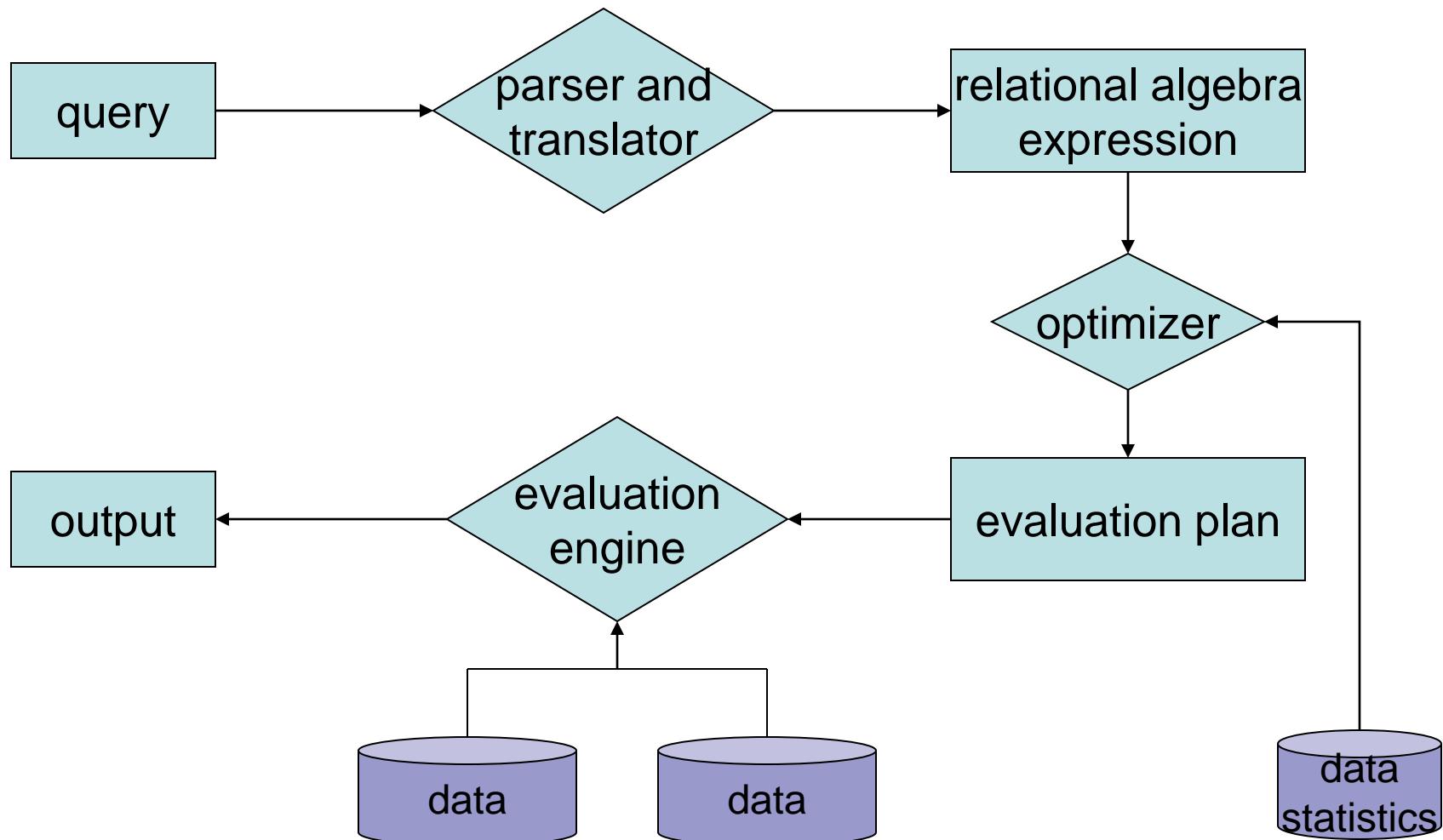
- SELECT * FROM student WHERE name=Paul
- Parse query and translate
 - check syntax, verify names, etc
 - translate into relational algebra (RDBMS)
 - create **evaluation plans**
- Find best plan (**optimization**)
- Execute plan

student	
<u>cid</u>	name
00112233	Paul
00112238	Rob
00112235	Matt

takes	
<u>cid</u>	<u>courseid</u>
00112233	312
00112233	395
00112235	312

course	
<u>courseid</u>	<u>coursename</u>
312	Advanced DBs
395	Machine Learning

Query Processing (2/2)



Relational Algebra (1/2)

- Query language
- Operations:
 - select: σ
 - project: π
 - union: \cup
 - difference: -
 - product: \times
 - join: \bowtie
- Extended relational algebra operations:

Relational Algebra (2/2)

- $\text{SELECT } * \text{ FROM student WHERE name=Paul}$
 - $\sigma_{\text{name}=\text{Paul}}(\text{student})$
- $\pi_{\text{name}}(\sigma_{\text{cid}<00112235}(\text{student}))$
- $\pi_{\text{name}}(\sigma_{\text{coursename}=\text{Advanced DBs}}((\text{student} \bowtie_{\text{cid}} \text{takes}) \bowtie_{\text{courseid}} \text{course}))$

student	
<u>cid</u>	<u>name</u>
00112233	Paul
00112238	Rob
00112235	Matt

takes	
<u>cid</u>	<u>courseid</u>
00112233	312
00112233	395
00112235	312

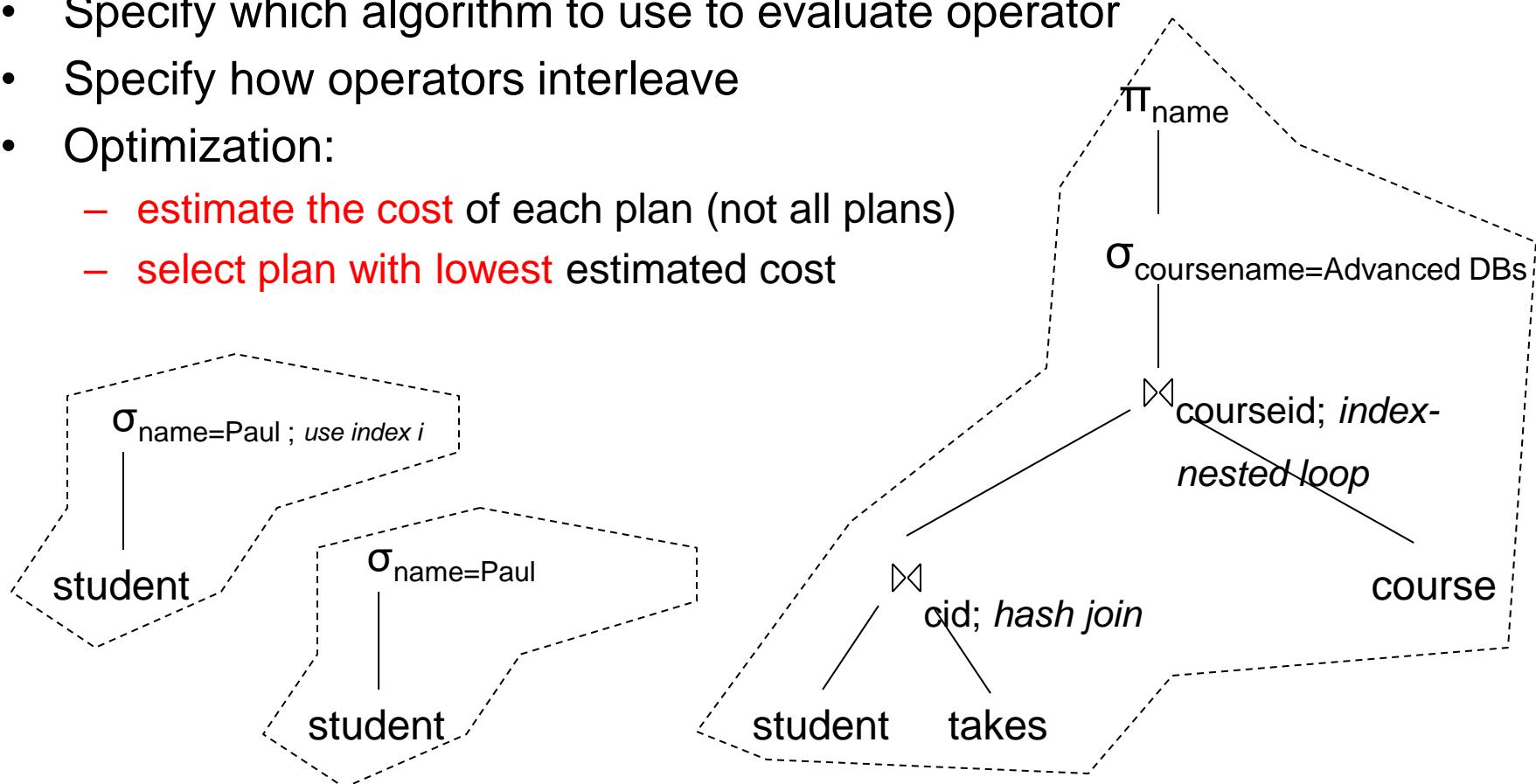
course	
<u>courseid</u>	<u>coursename</u>
312	Advanced DBs
395	Machine Learning

Why Optimize?

- Many alternative options to evaluate a query
 - $\pi_{\text{name}}(\sigma_{\text{coursename}=\text{Advanced DBs}}((\text{student} \Join_{\text{cid}} \text{takes}) \Join_{\text{courseid}} \text{course}))$
 - $\pi_{\text{name}}((\text{student} \Join_{\text{cid}} \text{takes}) \Join_{\text{courseid}} \sigma_{\text{coursename}=\text{Advanced DBs}}(\text{course}))$
- Several options to evaluate a single operation
 - $\sigma_{\text{name}=\text{Paul}}(\text{student})$
 - scan file
 - use secondary index on student.name
- Multiple access paths
 - access path: how can records be accessed (using index; which index?; etc.)

Evaluation plans

- Specify which access path to follow
- Specify which algorithm to use to evaluate operator
- Specify how operators interleave
- Optimization:
 - estimate the cost of each plan (not all plans)
 - select plan with lowest estimated cost



Estimating Cost

- What needs to be considered:
 - Disk I/Os
 - sequential (reading neighbouring pages is faster)
 - random
 - CPU time
 - Network communication
- What are we going to consider:
 - Disk I/Os
 - page (**data block**) reads/writes
 - Ignoring cost of writing **final output**

Operations and Costs (1/2)

- Operations: σ , π , \cup , \cap , \setminus , \times , \bowtie
- Costs:
 - N_R : number of records in R (other notation: T_R or $T(R)$ -> tuple)
 - L_R : size of record in R (length of record -> $L(R)$)
 - bf_R : blocking factor (other notation: F_R or $bf(R)$)
 - number of records in a page (datablock)
 - B_R : number of pages to store relation R (-> $B(R)$)
 - $V(R, A)$: number of distinct values of attribute A in R
other notation: $I_A(R)$ (Image size)
 - $SC(R, A)$: selection cardinality of A in R (number of matching rec.)
 - A key: $SC(R, A)=1$
 - A nonkey: $SC(R, A)=T_R / V(R, A)$ (uniform distribution assumption)
 - HT_i : number of levels in index I (-> height of tree)
 - rounding up fractions and logarithms

Operations and Costs (2/2)

- relation $takes(cid, name)$
 - 7000 tuples
 - student cid 8 bytes
 - course id 4 bytes $\rightarrow L(R) = 8+4 = 12$ bytes
 - 40 courses
 - 1000 students
 - page size 512 bytes
 - output size (in pages) of query:
which students take the Advanced DBs course?
 - $T_{takes} = 7000$
 - $V(courseid, takes) = 40$
 - $SC(courseid, takes) = \text{ceil}(T_{takes}/V(courseid, takes)) = \text{ceil}(7000/40) = 175$
 - $bf_{takes} = \text{floor}(512/12) = 42$ $B_{takes} = 7000/42 = 167$ pages
 - $bf_{output} = \text{floor}(512/8) = 64$ $B_{output} = 175/64 = 3$ pages

Cost of Selection σ (1/2)

- **Linear search**
 - read all pages, find records that match (assuming equality search)
 - **average cost:**
 - nonkey (multiple occurrences): B_R , key: $0.5 * B_R$ (half in average)
- **Binary search**
 - on ordered field
 - **average cost:** $\lceil \log_2 B_R \rceil + m$
 - m additional pages to be read (first found then read the **duplicates**)
 - $m = \text{ceil}(\text{SC}(A,R)/bf_R) - 1$
- **Primary/Clustered Index (B+ tree)**
 - **average cost:**
 - single record: $HT_i + 1$
 - multiple records: $HT_i + \text{ceil}(\text{SC}(R,A)/bf_R)$

Cost of Selection σ (2/2)

- Secondary Index (B+ tree)
 - average cost:
 - key field: $HT_i + 1$
 - nonkey field
 - worst case $HT_i + SC(A, R)$
 - linear search more desirable if many matching records !!!

Complex selection σ_{expr}

- **conjunctive selections:** $\sigma_{\theta_1 \wedge \theta_2 \dots \wedge \theta_n}$
 - perform simple selection using θ_i with the lowest evaluation cost
 - e.g. **using an index** corresponding to θ_i
 - apply remaining conditions θ on the resulting records
 - $\sigma_{cid > 00112233 \wedge courseid = 312}$ (*takes*)
 - cost: the cost of the simple selection on selected θ
 - **multiple indices**
 - select indices that correspond to θ_i s
 - scan indices and return RIDs (ROWID in Oracle)
 - answer: intersection of RIDs
 - cost: the sum of costs + record retrieval
- **disjunctive selections:** $\sigma_{\theta_1 \vee \theta_2 \dots \vee \theta_n}$
 - **multiple indices**
 - union of RIDs
 - **linear search**

Projection and set operations

- SELECT DISTINCT cid FROM takes
 - π requires **duplicate elimination**
 - **sorting**
- set operations require duplicate elimination
 - $R \cap S$
 - $R \cup S$
 - **sorting**

Sorting

- efficient evaluation for many operations
- required by query:
 - `SELECT cid, name FROM student ORDER BY name`
- implementations
 - **internal sorting** (if records fit in memory)
 - **external sorting**
 - (that's why we need temporary space on disk)

External Sort-Merge Algorithm (1/3)

- **Sort stage:** create sorted *runs*

$i=0;$

repeat

 read M pages of relation R into memory (**M : size of Memory**)

 sort the M pages

 write them into file R_i

 increment i

until no more pages

$N = i$ // number of runs

External Sort-Merge Algorithm (2/3)

- **Merge stage:** merge sorted *runs*

//assuming $N < M$ ($N \leq M-1$ we need 1 output buffer)

allocate a page for each run file R_i // N pages allocated

read a page P_i of each R_i

repeat

 choose first record (in sort order) among N pages, say from page P_j

 write record to output and delete from page P_j

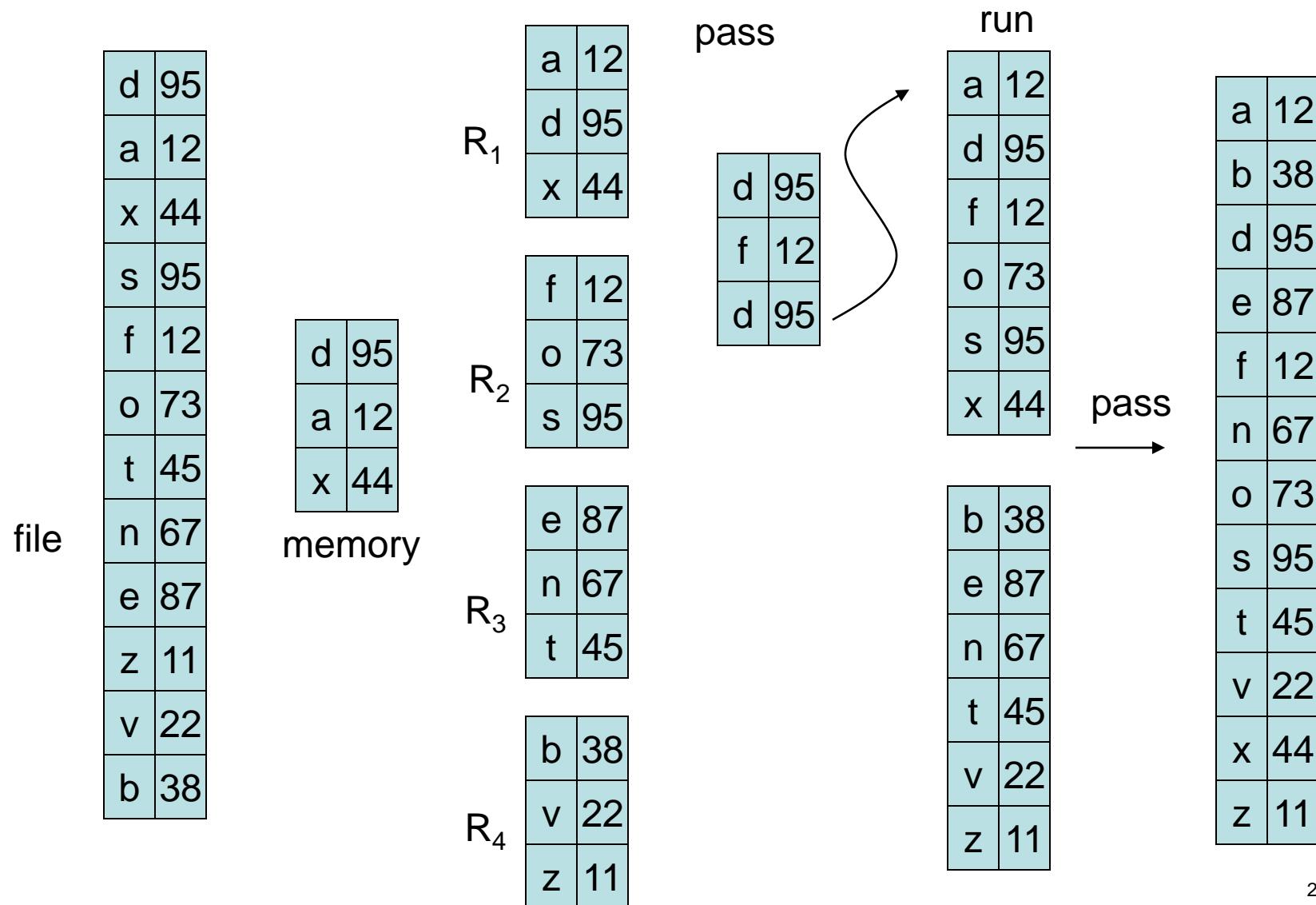
 if page is empty read next page P_j' from R_j

until all pages are empty

External Sort-Merge Algorithm (3/3)

- Merge stage: merge sorted *runs*
- What if $N \geq M$?
 - perform **multiple passes**
 - each *pass* merges $M-1$ runs until relation is processed
 - in next pass number of runs is reduced
 - final *pass* generated sorted output

Sort-Merge Example



Sort-Merge cost

- B_R the number of pages of R
- Sort stage: $2 * B_R$
 - read/write relation
- Merge stage:
 - initially $\left\lceil \frac{B_R}{M} \right\rceil$ runs to be merged
 - each pass $M-1$ runs sorted
 - thus, total number of passes: $\left\lceil \log_{M-1} \left(\frac{B_R}{M} \right) \right\rceil$
 - at each pass $2 * B_R$ pages are read/written
 - read/write relation ($B_R + B_R$)
 - apart from final write (B_R)
- Total cost:
 - $2 * B_R + 2 * B_R * \left\lceil \log_{M-1} \left(\frac{B_R}{M} \right) \right\rceil - B_R$ eg. $B_R = 1000000, M=100$

Projection

- $\pi_{A1, A2\dots}(R)$
- remove unwanted attributes
 - scan and drop attributes
- remove duplicate records
 - **sort resulting records** using all attributes as sort order
 - scan sorted result, **eliminate duplicates** (adjacent)
- cost
 - initial scan + **sorting** + final scan

Join

- $\Pi_{\text{name}}(\sigma_{\text{coursetitle}=\text{Advanced DBs}}((\text{student} \bowtie_{\text{cid}} \text{takes}) \bowtie_{\text{courseid}} \text{course}))$
- implementations
 - nested loop join
 - block-nested loop join
 - indexed nested loop join
 - sort-merge join
 - hash join

Nested loop join (1/2)

- $R \bowtie S$

for each tuple t_R of R

 for each t_S of S

 if (t_R t_S match) output $t_R.t_S$

 end

end

- Works for any join condition
- S inner relation
- R outer relation

Nested loop join (2/2)

- Costs:
 - **best case** when smaller relation fits in memory
 - use it as inner relation
 - $B_R + B_S$
 - **worst case** when memory holds one page of each relation
 - S scanned for each tuple in R
 - $T_R * B_S + B_R$

Block nested loop join (1/2)

for each page X_R of R

 for each page X_S of S

 for each tuple t_R in X_R

 for each t_S in X_S

 if (t_R t_S match) output $t_R.t_S$

 end

 end

 end

end

Block nested loop join (2/2)

- Costs:
 - **best case** when smaller relation fits in memory
 - use it as inner relation
 - $B_R + B_S$
 - **worst case** when memory holds one page of each relation
 - S scanned for each page in R
 - $B_R * B_S + B_R$

Block nested loop join (an improvement)

Memory size: M

for each $M - 1$ page size chunk M_R of R

 for each page X_S of S

 for each tuple t_R in M_R

 for each t_S in X_S

 if (t_R t_S match) output $t_R.t_S$

 end

 end

 end

end

Block nested loop join (an improvement)

- Costs:
 - **best case** when smaller relation fits in memory
 - use it as inner relation
 - $B_R + B_S$
 - **general case**
 - S scanned for each $M-1$ size chunk in R
 - $(B_R / (M-1)) * B_S + B_R$

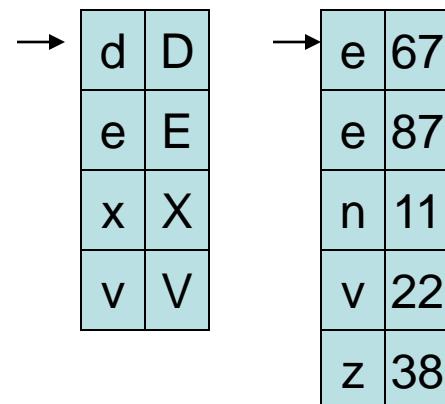
Indexed nested loop join

- $R \bowtie S$
- Index on inner relation (S)
- for each tuple in outer relation (R) *probe* index of inner relation
- Costs:
 - $B_R + T_R * c$
 - c the cost of index-based selection of inner relation
 - $c \approx T(s)/V(s,A)$

(if A is the join column and index is kept in memory)
 - relation with fewer records as outer relation

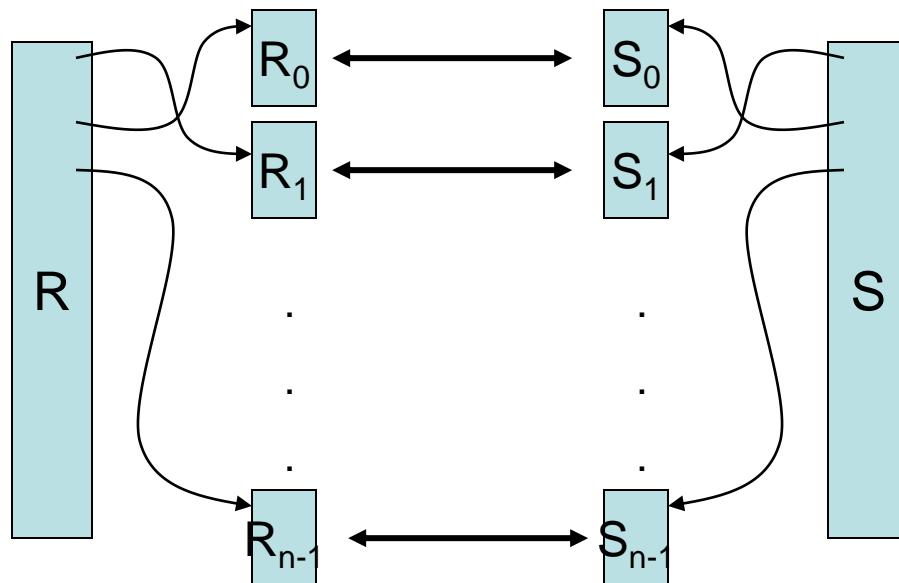
Sort-merge join

- $R \bowtie S$
- Relations sorted on the join attribute
- Merge sorted relations
 - pointers to first record in each relation
 - read in a group of records of S with the same values in the join attribute
 - read records of R and process
- Relations in sorted order to be read once
- Cost:
 - cost of sorting + $B_S + B_R$



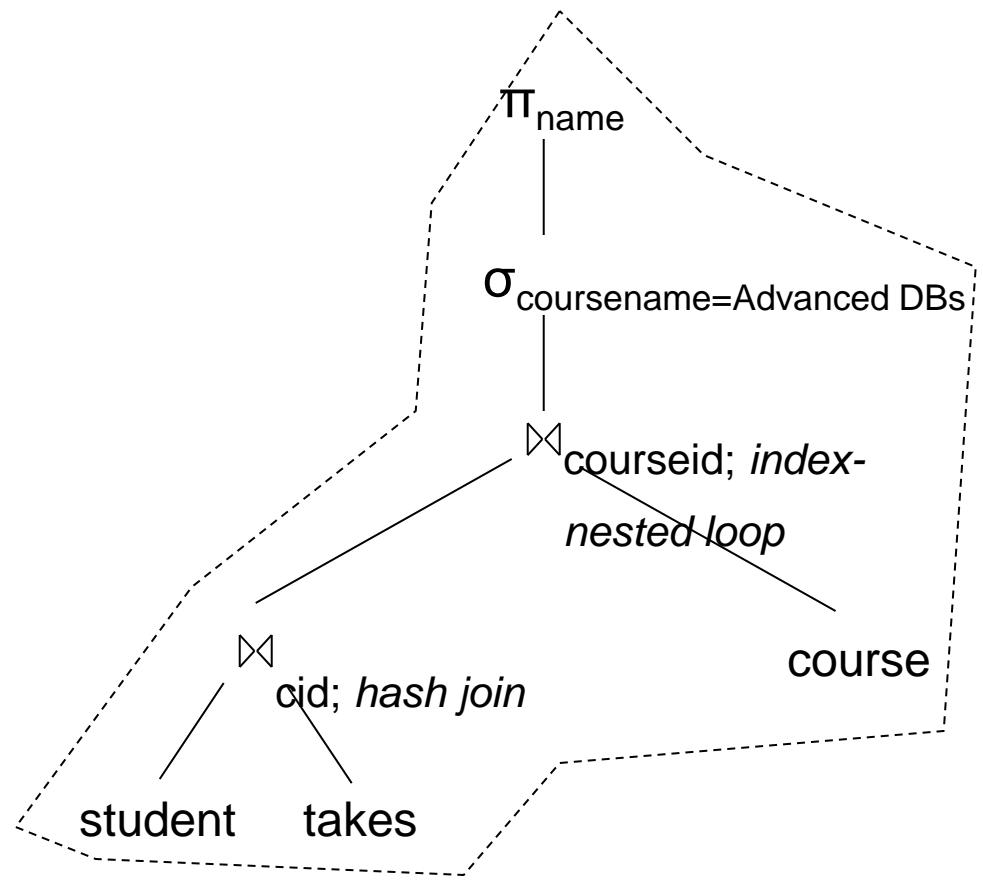
Hash join

- $R \bowtie S$
- use h_1 on joining attribute to map records to partitions that fit in memory
 - records of R are partitioned into $R_0 \dots R_{n-1}$
 - records of S are partitioned into $S_0 \dots S_{n-1}$
- join records in corresponding partitions
 - using a hash-based indexed block nested loop join
- Cost: $2*(B_R+B_S) + (B_R+B_S)$



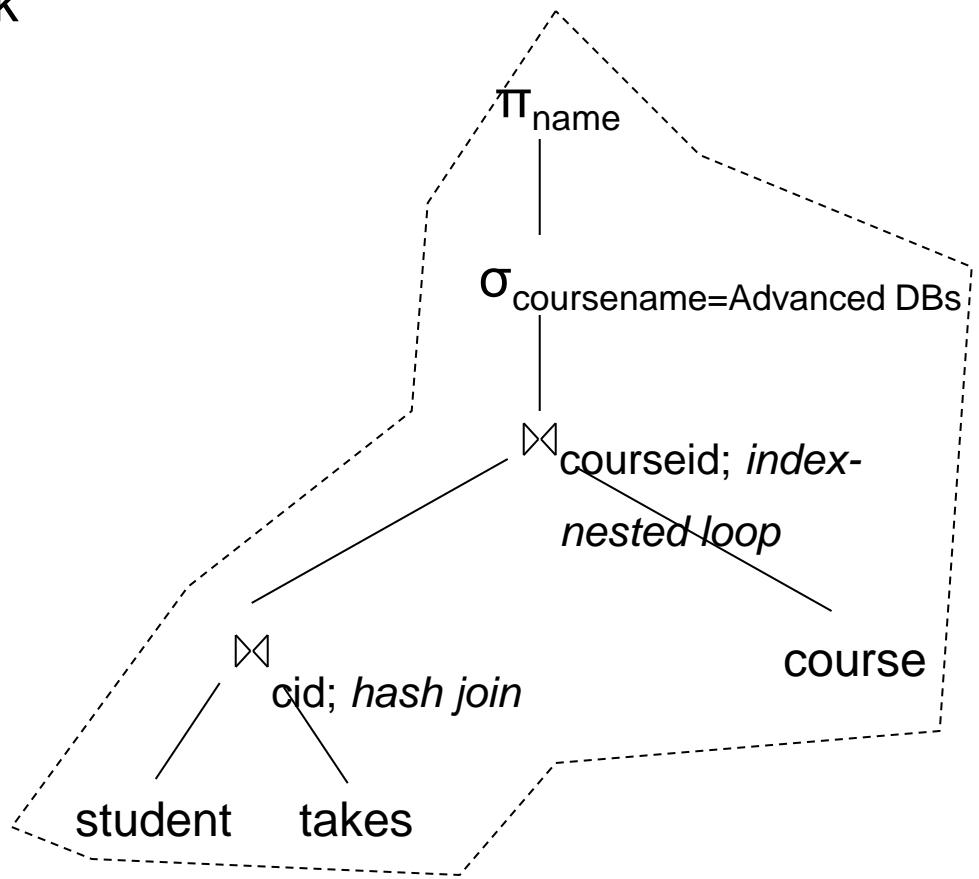
Evaluation

- evaluate multiple operations in a plan
- materialization
- pipelining



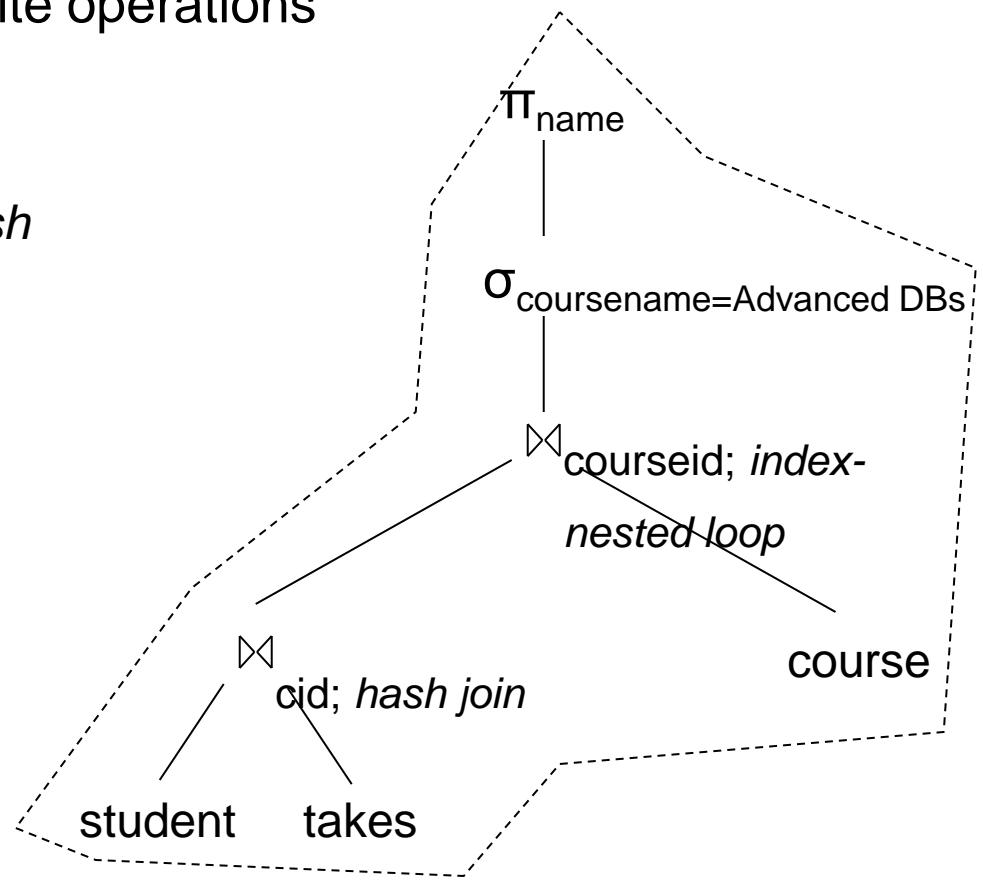
Materialization

- **create** and read **temporary relations**
- create implies writing to disk
 - more page writes



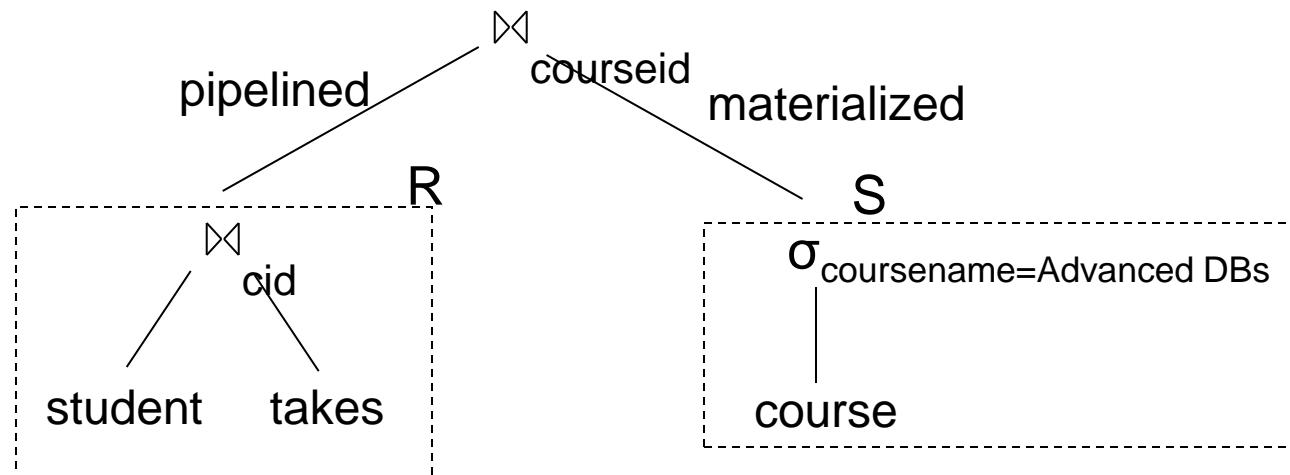
Pipelining (1/2)

- creating a pipeline of operations
- reduces number of read-write operations
- implementations
 - demand-driven - data *pull*
 - producer-driven - data *push*



Pipelining (2/2)

- can pipelining always be used?
- any algorithm?
- cost of $R \bowtie S$
 - materialization and hash join: $B_R + 3(B_R + B_S)$
 - pipelining and indexed nested loop join: $T_R * HT_i$

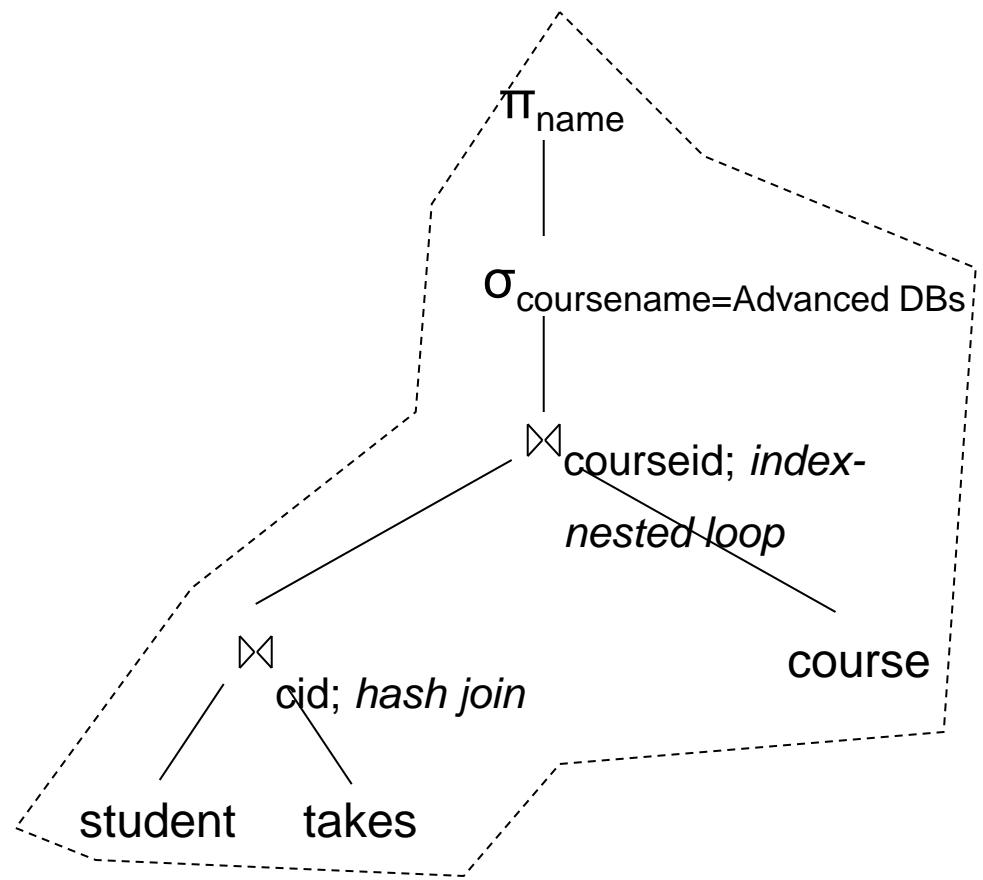


Choosing evaluation plans

- cost based optimization
- enumeration of plans
 - $R \bowtie S \bowtie T$, 12 possible orders $(3! * 2)$
 $(R \bowtie S) \bowtie T$, $R \bowtie (S \bowtie T)$
- cost estimation of each plan
- overall cost
 - cannot optimize operation independently

Cost estimation

- operation (σ , π , \bowtie ...)
- implementation
- size of inputs
- size of outputs
- sorting

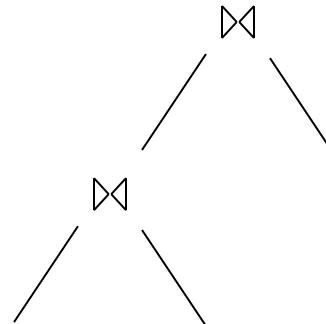


Expression Equivalence

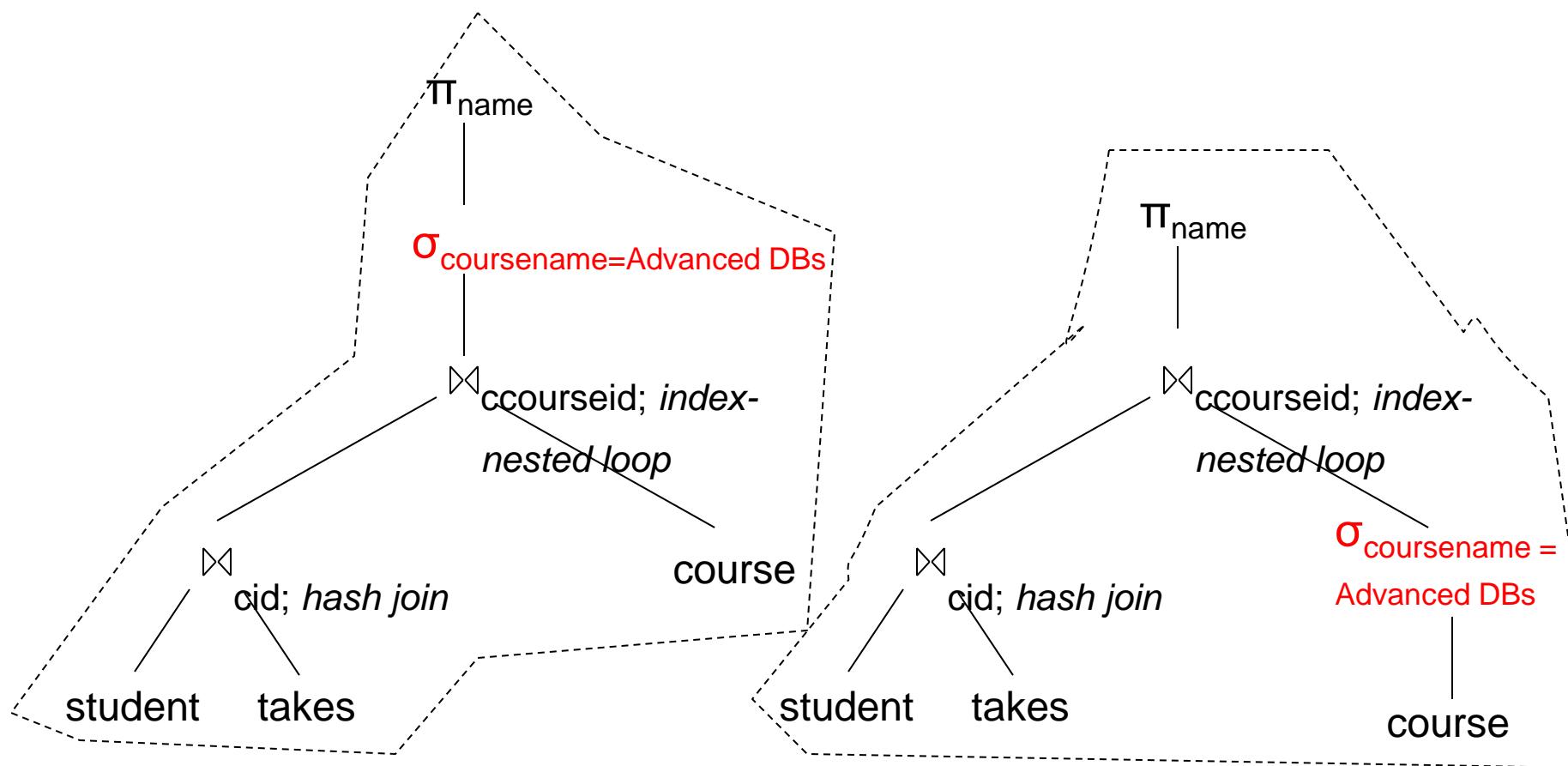
- conjunctive selection decomposition
 - $\sigma_{\theta_1 \wedge \theta_2}(R) = \sigma_{\theta_1}(\sigma_{\theta_2}(R))$
- commutativity of selection
 - $\sigma_{\theta_1}(\sigma_{\theta_2}(R)) = \sigma_{\theta_2}(\sigma_{\theta_1}(R))$
- combining selection with join and product
 - $\sigma_{\theta_1}(R \times S) = \dots \quad R \bowtie_{\theta_1} S = \dots$
- commutativity of joins
 - $R \bowtie_{\theta_1} S = S \bowtie_{\theta_1} R$
- distribution of selection over join
 - $\sigma_{\theta_1 \wedge \theta_2}(R \bowtie S) = \sigma_{\theta_1}(R) \bowtie \sigma_{\theta_2}(S)$
- distribution of projection over join
 - $\pi_{A1, A2}(R \bowtie S) = \pi_{A1}(R) \bowtie \pi_{A2}(S)$
- associativity of joins: $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$

Cost Optimizer (1/2)

- transforms expressions
 - equivalent expressions
 - heuristics, *rules of thumb*
 - perform **selections early**
 - perform **projections early**
 - replace products followed by selection σ ($R \times S$) with joins $R \bowtie S$
 - start with joins, selections with smallest result
 - create **left-deep join trees**



Cost Optimizer (2/2)



Summary

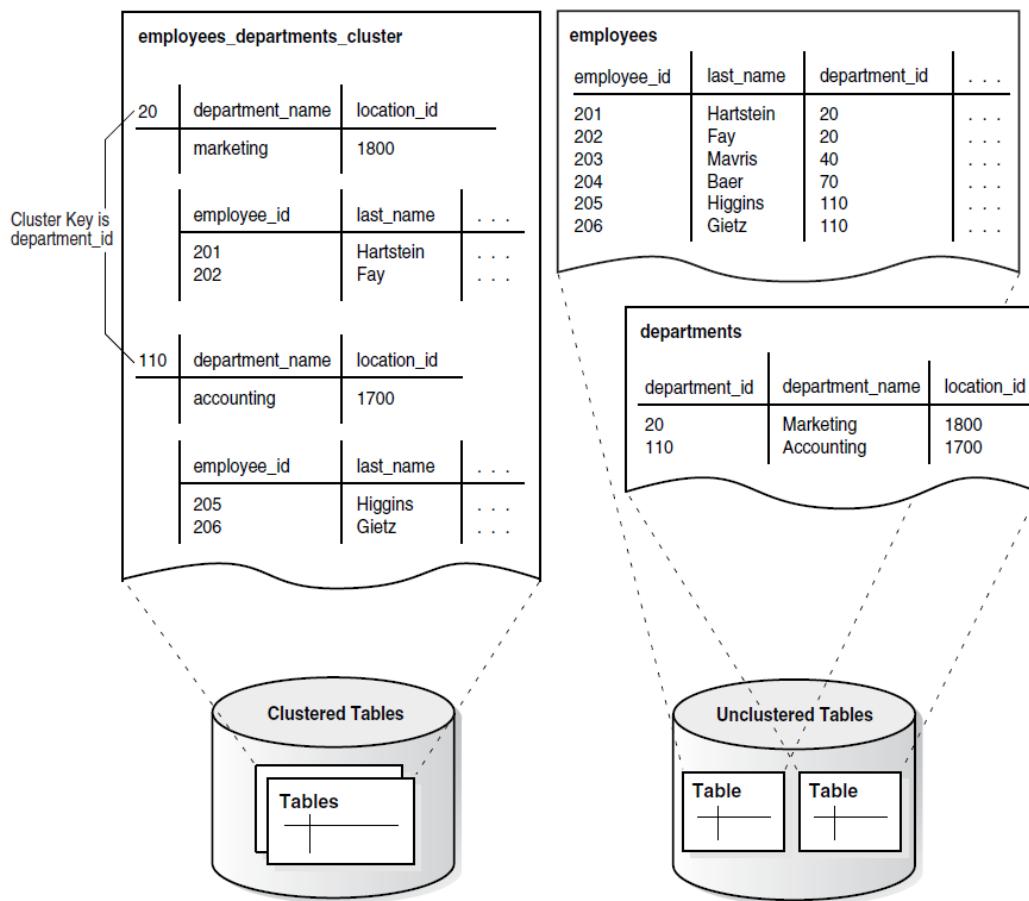
- Estimating the cost of a single operation
- Estimating the cost of a query plan
- Optimization
 - choose the most efficient plan

Oracle database concepts

A **table cluster** is a group of tables that share common columns and store related data in the same blocks. When tables are clustered, a single data block can contain rows from multiple tables. For example, a block can store rows from both the `employees` and `departments` tables rather than from only a single table.

The cluster key is the column or columns that the clustered tables have in common. For example, the `employees` and `departments` tables share the `department_id` column. You specify the cluster key when creating the table cluster and when creating every table added to the table cluster.

Figure 2–6 Clustered Table Data



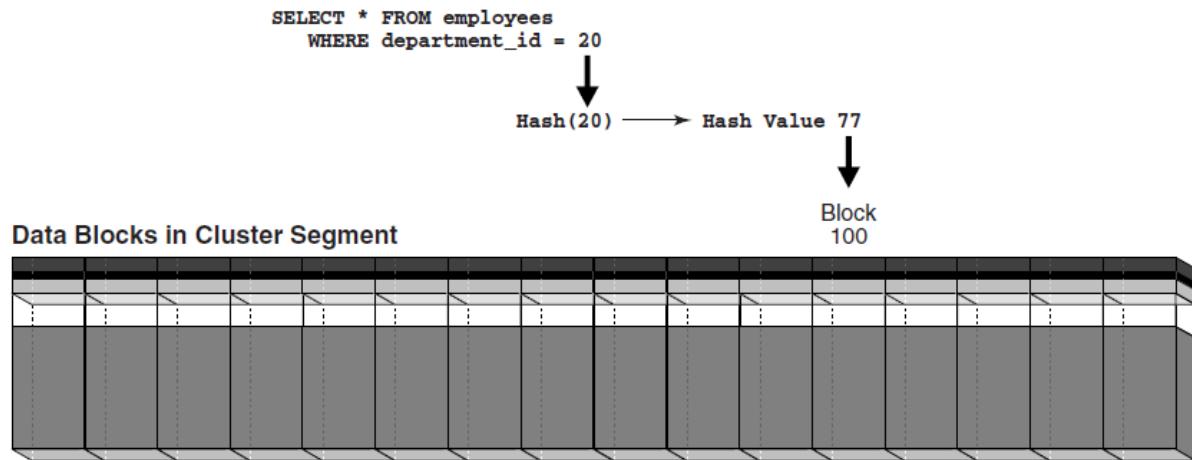
A **hash cluster** is like an indexed cluster, except the **index key** is replaced with a **hash function**. No separate cluster index exists. In a hash cluster, the data is the index.

With an indexed table or indexed cluster, **Oracle Database locates table rows using key values** stored in a separate index. To find or store a row in an indexed table or table cluster, the database must perform at least two I/Os:

- One or more I/Os to find or store the key value in the index
- Another I/O to read or write the row in the table or table cluster

To find or store a row in a hash cluster, Oracle Database applies the hash function to the cluster key value of the row. **The resulting hash value corresponds to a data block** in the cluster, which the database reads or writes on behalf of the issued statement.

Figure 2–7 Retrieving Data from a Hash Cluster



Partitions

In an Oracle database, partitioning enables you to decompose **very large tables** and **indexes** into **smaller** and more manageable **pieces** called partitions. Each partition is an independent object with its own name and optionally its own storage characteristics. A partitioned object has pieces that can be managed either collectively or individually. DDL statements can manipulate partitions rather than entire tables or indexes.

Each partition of a table or index must have the same logical attributes, such as column names, data types, and constraints. For example, all partitions in a table share the same column and constraint definitions. However, each partition can have separate physical attributes, such as the tablespace to which it belongs.

The **partition key** is a set of one or more columns that determines the partition in which each row in a partitioned table should go. Each row is unambiguously assigned to a single partition.

Oracle Partitioning offers several partitioning strategies that control how the database places data into partitions. The basic strategies are **range, list, and hash partitioning**.

A **single-level partitioning** uses only one method of data distribution, for example, only list partitioning or only range partitioning.

In **composite partitioning**, a table is partitioned by one data distribution method and then each partition is further divided into **subpartitions** using a second data distribution method.

In **range partitioning**, the database maps rows to partitions based on ranges of values of the partitioning key. The range partition key value determines the non-inclusive high bound for a specified partition.

In **list partitioning**, the database uses a list of discrete values as the partition key for each partition. You can use list partitioning to control how individual rows map to specific partitions. By using lists, you can group and organize related sets of data when the key used to identify them is not conveniently ordered.

In **hash partitioning**, the database maps rows to partitions based on a hashing algorithm that the database applies to the user-specified partitioning key.

A partitioned table is made up of one or more **table partition segments**. If you create a partitioned table named `hash_products`, then **no table segment** is allocated for this table. Instead, the database

stores data for each table partition in its own **partition segment**. Each table partition segment contains a portion of the table data.

Figure 4-1 Range Partitions

The figure displays three separate tables, each representing a range partition of the SALES table for a specific year. Each table has the same structure: PROD_ID, CUST_ID, TIME_ID, CHANNEL_ID, PROMO_ID, QUANTITY_SOLD, and AMOUNT_SOLD.

Table Partition SALES_1998						
PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
40	100530	30-NOV-98	9	33	1	44.99
125	9417	04-FEB-98	3	999	1	16.86
45	9491	28-AUG-98	4	350	1	47.45

Table Partition SALES_1999						
PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
116	11393	05-JUN-99	2	999	1	12.18
36	4523	27-JAN-99	3	999	1	53.89
24	11899	26-JUN-99	4	999	1	43.04

Table Partition SALES_2000						
PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
133	9450	01-DEC-00	2	999	1	31.28
35	2606	17-FEB-00	3	999	1	54.94

Range partitions based on TIME_ID column.

A **partitioned index** is an index that, like a partitioned table, has been divided into smaller and more manageable pieces.

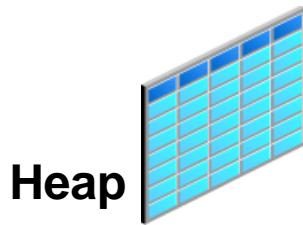
Global indexes are partitioned independently of the table on which they are created, whereas **local** indexes are automatically linked to the partitioning method for a table. Like partitioned tables, partitioned indexes improve manageability, availability, performance, and scalability.

In a **local partitioned index**, the index is partitioned on the same columns, with the same number of partitions and the same partition bounds as its table.

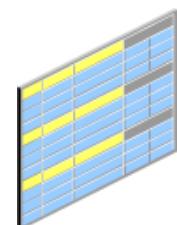
Local partitioned indexes are either **prefixed** or **nonprefixed**.

In the prefixed case, the partition keys are on the leading edge of the index definition.

Table Types

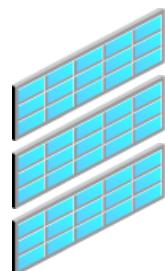


Heap

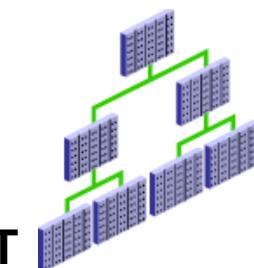


Clustered

• Type	• Description
• Ordinary (heap-organized) table	• Data is stored as an unordered collection (heap).
• Partitioned table	• Data is divided into smaller, more manageable pieces.
• Index-organized table (IOT)	• Data (including non-key values) is sorted and stored in a B-tree index structure.
• Clustered table	• Related data from more than one table are stored together.



Partitioned

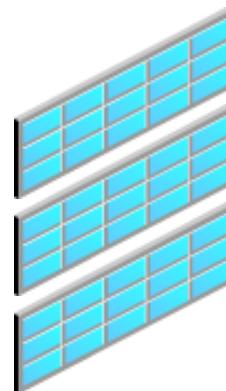


IOT

What Is a Partition and Why Use It?

- A partition is:

- A piece of a “very large” table or index
- Stored in its own segment
- Used for improved performance and manageability



RANGE PARTITION

```
CREATE TABLE eladasok ( szla_szam  NUMBER(5),
                        szla_nev   CHAR(30),
                        mennyiseg  NUMBER(6),
                        het        INTEGER )
PARTITION BY RANGE ( het )
(PARTITION negyedev1  VALUES LESS THAN ( 13 )
  TABLESPACE users,
PARTITION negyedev2  VALUES LESS THAN ( 26 )
  TABLESPACE example,
PARTITION negyedev3  VALUES LESS THAN ( 39 )
  TABLESPACE users )
```

DBA_PART_TABLES
DBA_TAB_PARTITIONS
DBA_TAB_SUBPARTITIONS

HASH PARTITION, LIST PARTITION

```
CREATE TABLE eladasok2 (szla_szam  NUMBER(5),
                        szla_nev   CHAR(30),
                        mennyiseg  NUMBER(6),
                        het        INTEGER )
PARTITION BY HASH ( het )
(PARTITION part1 TABLESPACE users,
  PARTITION part2 TABLESPACE example,
  PARTITION part3 TABLESPACE users );
```

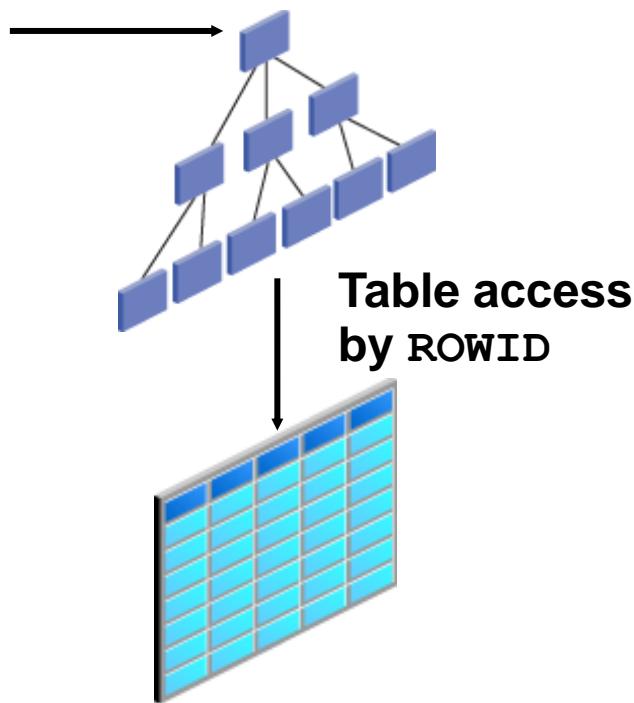
```
CREATE TABLE eladasok3 (szla_szam  NUMBER(5),
                        szla_nev   CHAR(30),
                        mennyiseg  NUMBER(6),
                        het        INTEGER )
PARTITION BY LIST ( het )
(PARTITION part1 VALUES(1,2,3,4,5) TABLESPACE users,
  PARTITION part2 VALUES(6,7,8,9) TABLESPACE example,
  PARTITION part3 VALUES(10,11,12,13) TABLESPACE users ) ;
```

SUBPARTITIONS (RANGE-HASH)

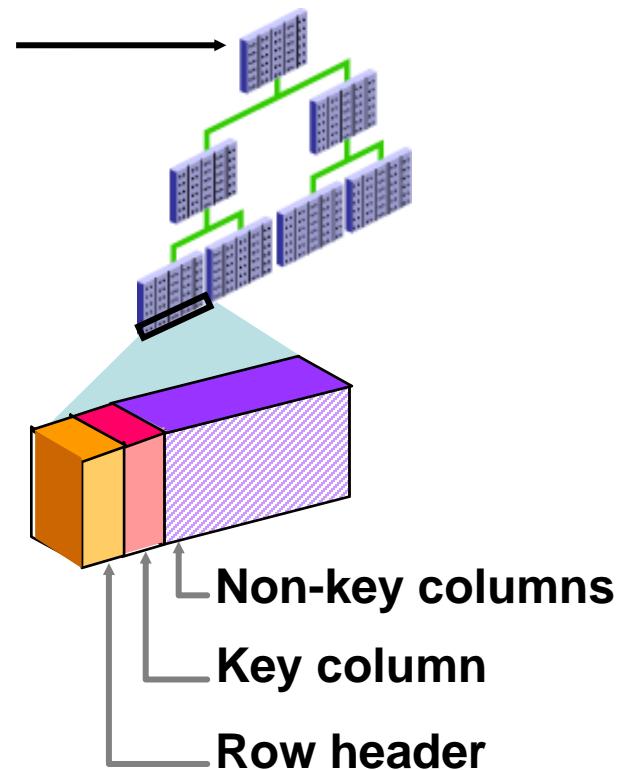
```
CREATE TABLE eladasok4 (szla_szam  NUMBER(5),
                       szla_nev   CHAR(30),
                       mennyiseg  NUMBER(6),
                       het        INTEGER )
PARTITION BY RANGE ( het )
SUBPARTITION BY HASH (mennyiseg)
SUBPARTITIONS 3
(PARTITION negyedev1 VALUES LESS THAN ( 13 )
 TABLESPACE users,
PARTITION negyedev2 VALUES LESS THAN ( 26 )
 TABLESPACE example,
PARTITION negyedev3 VALUES LESS THAN ( 39 )
 TABLESPACE users );
```

Index-Organized Tables

- Regular table access



IOT access



Index-Organized Tables and Heap Tables

- Compared to heap tables, IOTs:
 - Have **faster key-based access** to table data
 - **Do not duplicate** the storage of **primary key values**
 - Require less storage
 - Use **secondary indexes** and logical row IDs
 - Have higher availability because table reorganization does not invalidate secondary indexes

Index-Organized Tables

```
CREATE TABLE cikk_iot
( ckod integer,
  cnev varchar2(20),
  szin varchar2(15),
  suly float,
  CONSTRAINT cikk_iot_pk PRIMARY KEY (ckod) )
```

ORGANIZATION INDEX

```
PCTTHRESHOLD 20 INCLUDING cnev
OVERFLOW TABLESPACE users;
```

DBA_INDEXES index_type → 'IOT-TOP' table_name → 'CIKK_IOT'
DBA_TABLES.IOT_TYPE → 'IOT' or 'IOT_OVERFLOW'
DBA_TABLES.IOT_NAME → 'CIKK_IOT' for overflow segment

Clusters

ORD_NO	PROD	QTY	...
101	A4102	20	
102	A2091	11	
102	G7830	20	
102	N9587	26	
101	A5675	19	
101	W0824	10	

ORD_NO	ORD_DT	CUST_CD
101	05-JAN-97	R01
102	07-JAN-97	N45



**Unclustered orders and
order_item tables**

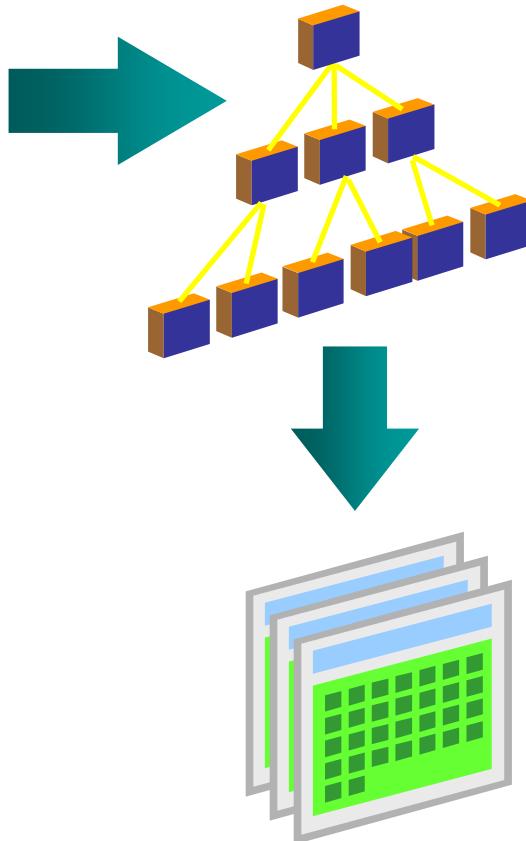
Cluster Key (ORD_NO)			
101	ORD DT	CUST CD	
05-JAN-97		R01	
PROD	QTY		
A4102	20		
A5675	19		
W0824	10		
102	ORD DT	CUST CD	
07-JAN-97		N45	
PROD	QTY		
A2091	11		
G7830	20		
N9587	26		



**Clustered orders and
order_item tables**

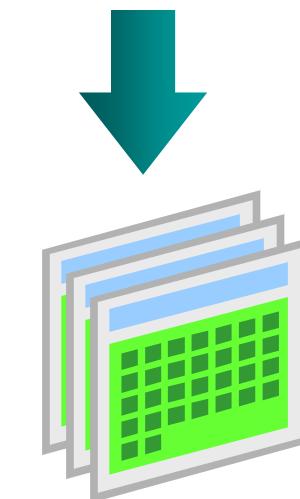
Cluster Types

Index cluster



Hash cluster

Hash function



Situations Where Clusters Are Useful

Criterion	Index	Hash
Uniform key distribution	X	X
Evenly spread key values		X
Rarely updated key	X	X
Often joined master-detail tables	X	
Predictable number of key values		X
Queries using equality predicate on key		X

INDEX CLUSTER

```
CREATE CLUSTER personnel
  ( department_number NUMBER(2) )  SIZE 512;
CREATE TABLE emp_cl
  ( empno NUMBER PRIMARY KEY,ename VARCHAR2(30),
    job VARCHAR2(27), mgr NUMBER(4), hiredate DATE,
    sal NUMBER(7,2), comm NUMBER(7,2),
    deptno NUMBER(2) NOT NULL)
CLUSTER personnel (deptno);
CREATE TABLE dept_cl
  ( deptno NUMBER(2), dname VARCHAR2(9), loc VARCHAR2(9))
CLUSTER personnel (deptno);
CREATE INDEX idx_personnel ON CLUSTER personnel;
```

DBA_CLUSTERS

DBA_CLU_COLUMNS

DBA_TABLES.CLUSTER_NAME → 'PERSONNEL'

HASH CLUSTER

```
CREATE CLUSTER personnel1
( department_number NUMBER )
SIZE 512 HASHKEYS 500
STORAGE (INITIAL 100K NEXT 50K);
```

```
CREATE CLUSTER personnel2
( home_area_code NUMBER, home_prefix NUMBER )
HASHKEYS 20
HASH IS MOD(home_area_code + home_prefix, 101);
```

```
CREATE CLUSTER personnel3
(deptno NUMBER)
SIZE 512 SINGLE TABLE HASHKEYS 500;
```

DBA_CLUSTERS
DBA_CLU_COLUMNS
DBA_CLUSTER_HASH_EXPRESSIONS

Ullman et al. :
Database System Principles

Notes 6: Query Processing

Query Processing

$Q \rightarrow \text{Query Plan}$

Focus: Relational System

- Others?

Example

Select B,D

From R,S

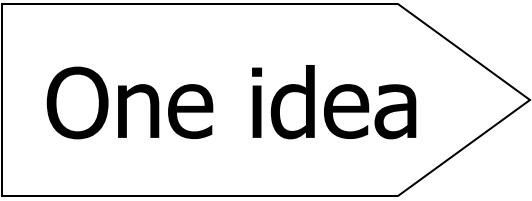
Where R.A = "c" \wedge S.E = 2 \wedge R.C=S.C

R	A	B	C	S	C	D	E
a	1	10		10	x	2	
b	1	20		20	y	2	
c	2	10		30	z	2	
d	2	35		40	x	1	
e	3	45		50	y	3	

Answer

B	D
2	x

- How do we execute query?



One idea

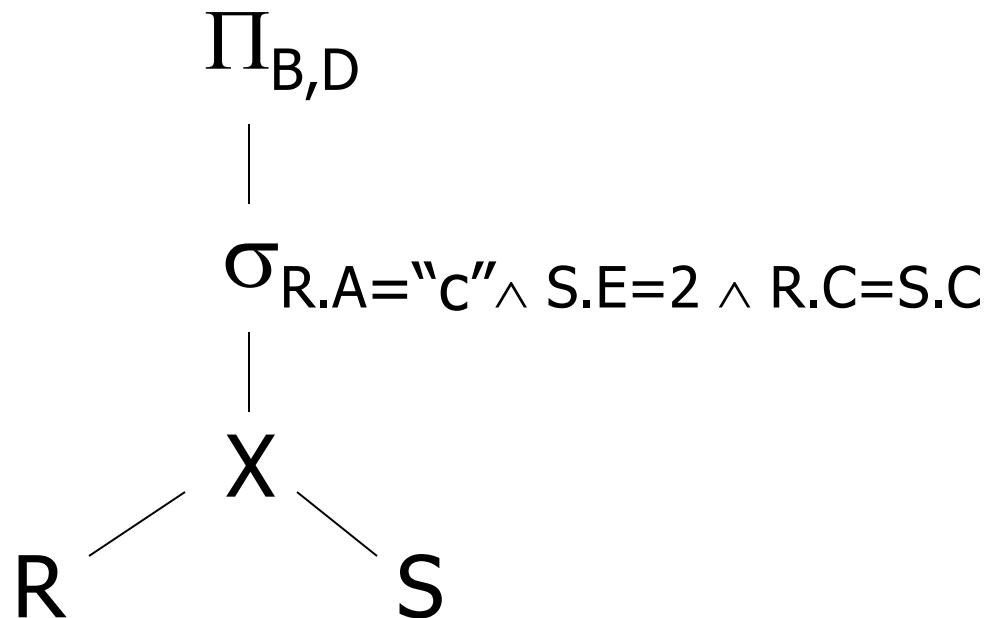
- Do Cartesian product
- Select tuples
- Do projection

RXS

	R.A	R.B	R.C	S.C	S.D	S.E
	a	1	10	10	x	2
	a	1	10	20	y	2
	.					
	.					
Bingo!	C	2	10	10	x	2
Got one...	.					
	.					

Relational Algebra - can be used to describe plans...

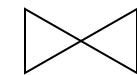
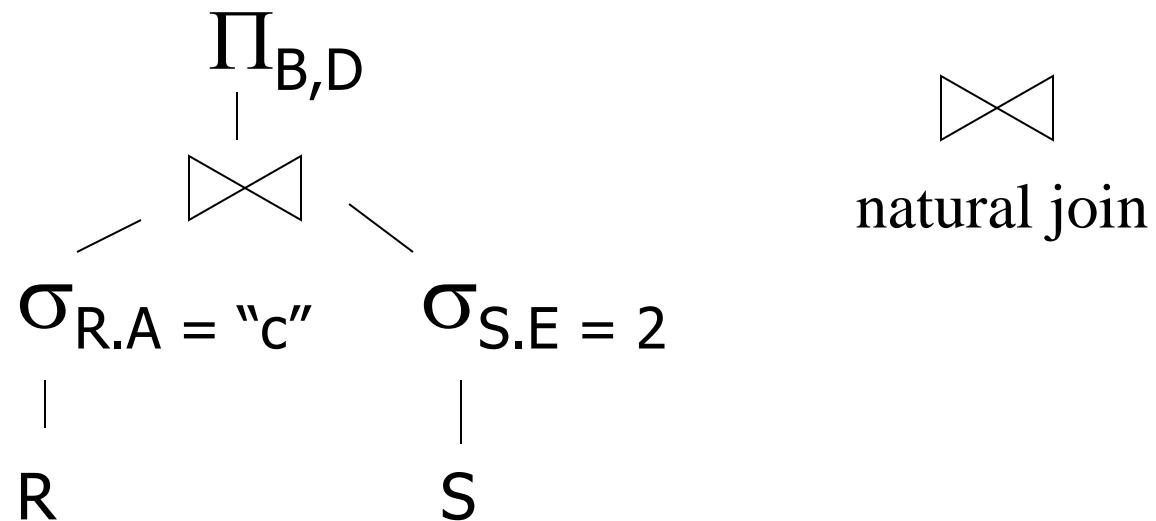
Ex: Plan I



OR: $\Pi_{B,D} [\sigma_{R.A='c' \wedge S.E=2 \wedge R.C = S.C} (R \times S)]$

Another idea:

Plan II



natural join

R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45

$\sigma(R)$

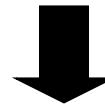
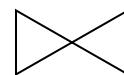
A	B	C
c	2	10

$\sigma(S)$

C	D	E
10	x	2
20	y	2
30	z	2

S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3



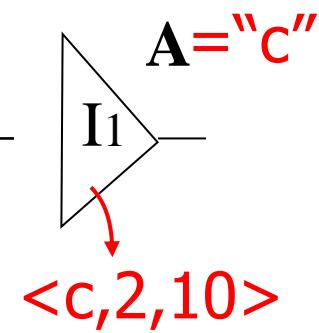
Plan III

Use R.A and S.C Indexes

- (1) Use R.A index to select R tuples
with R.A = "c"
- (2) For each R.C value found, use S.C
index to find matching tuples
- (3) Eliminate S tuples S.E \neq 2
- (4) Join matching R,S tuples, project
B,D attributes and place in result

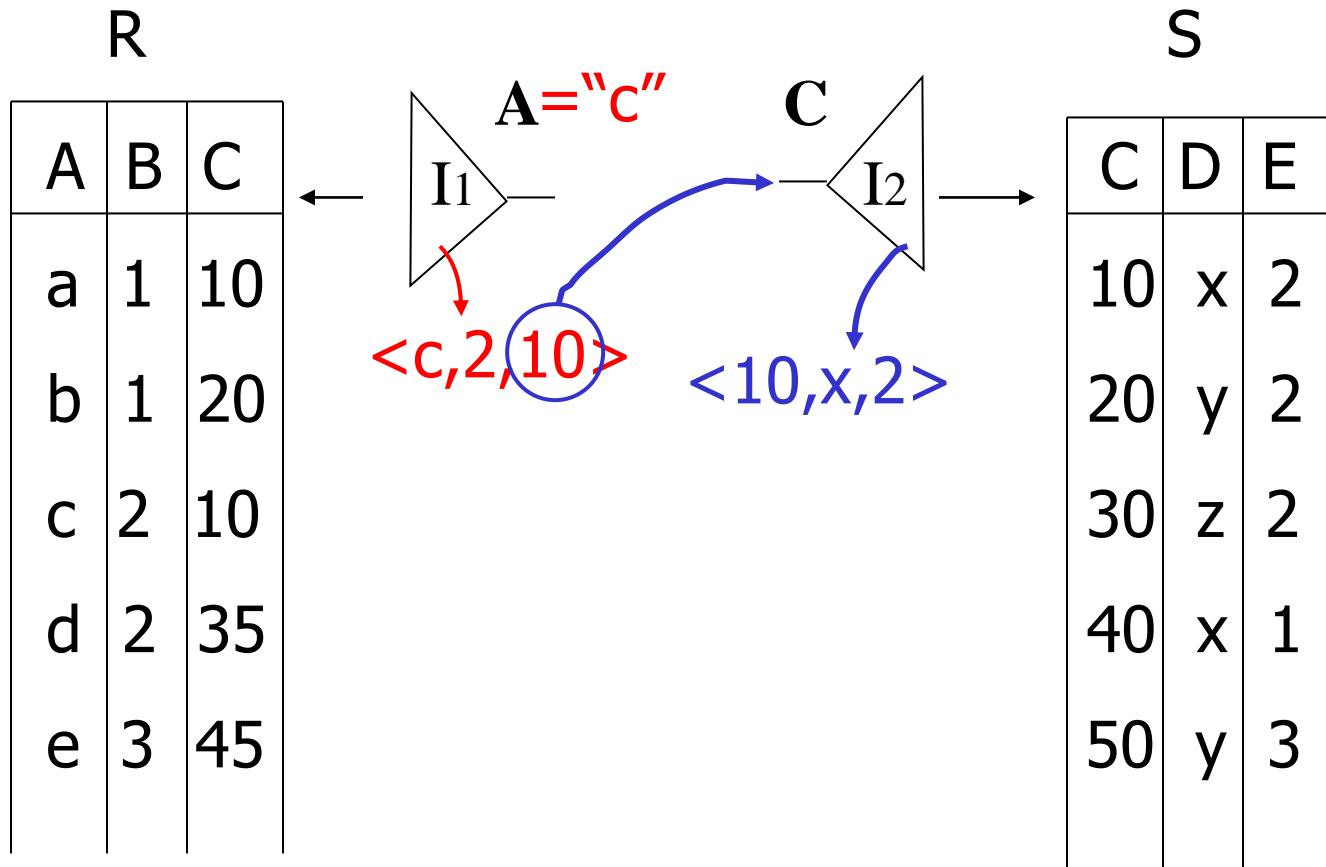
R

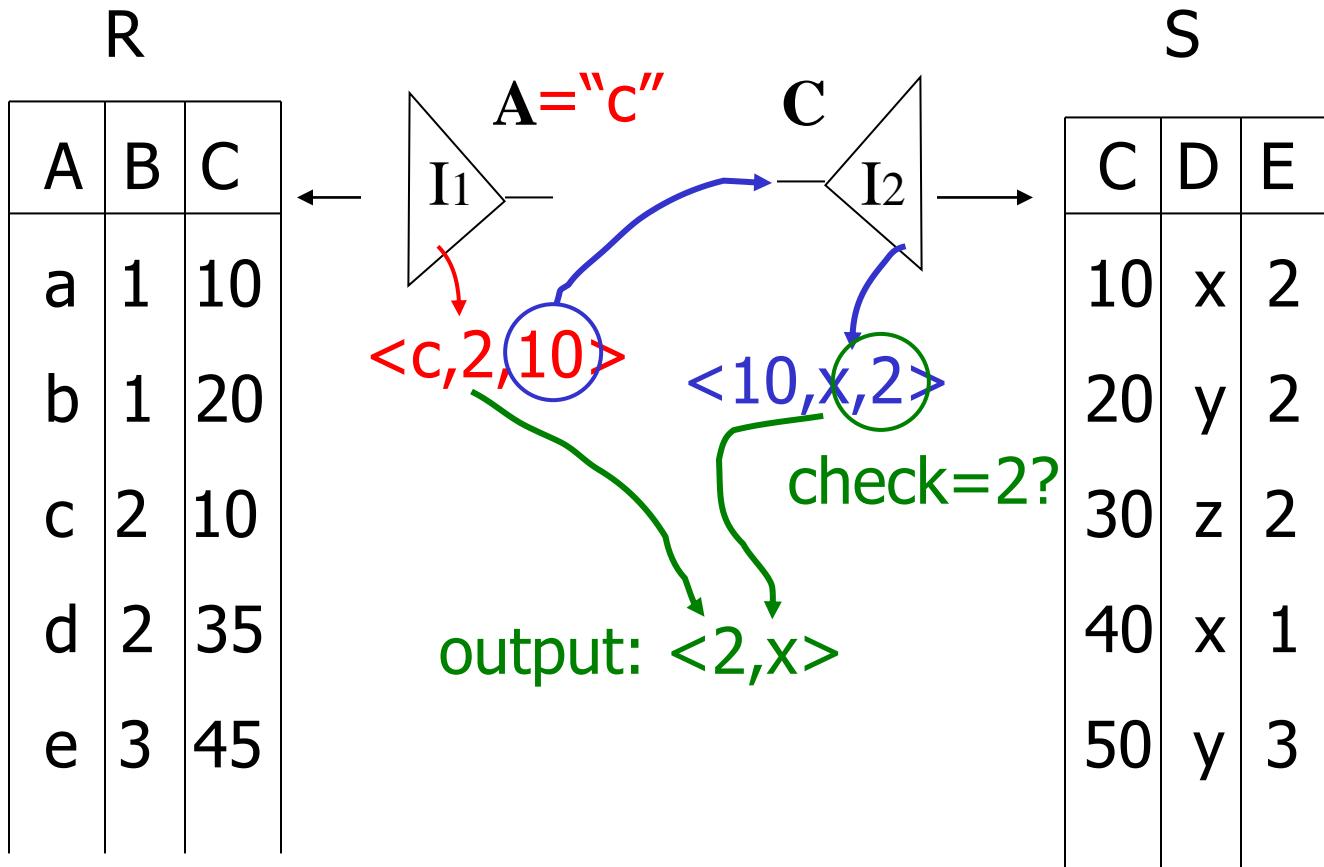
A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45



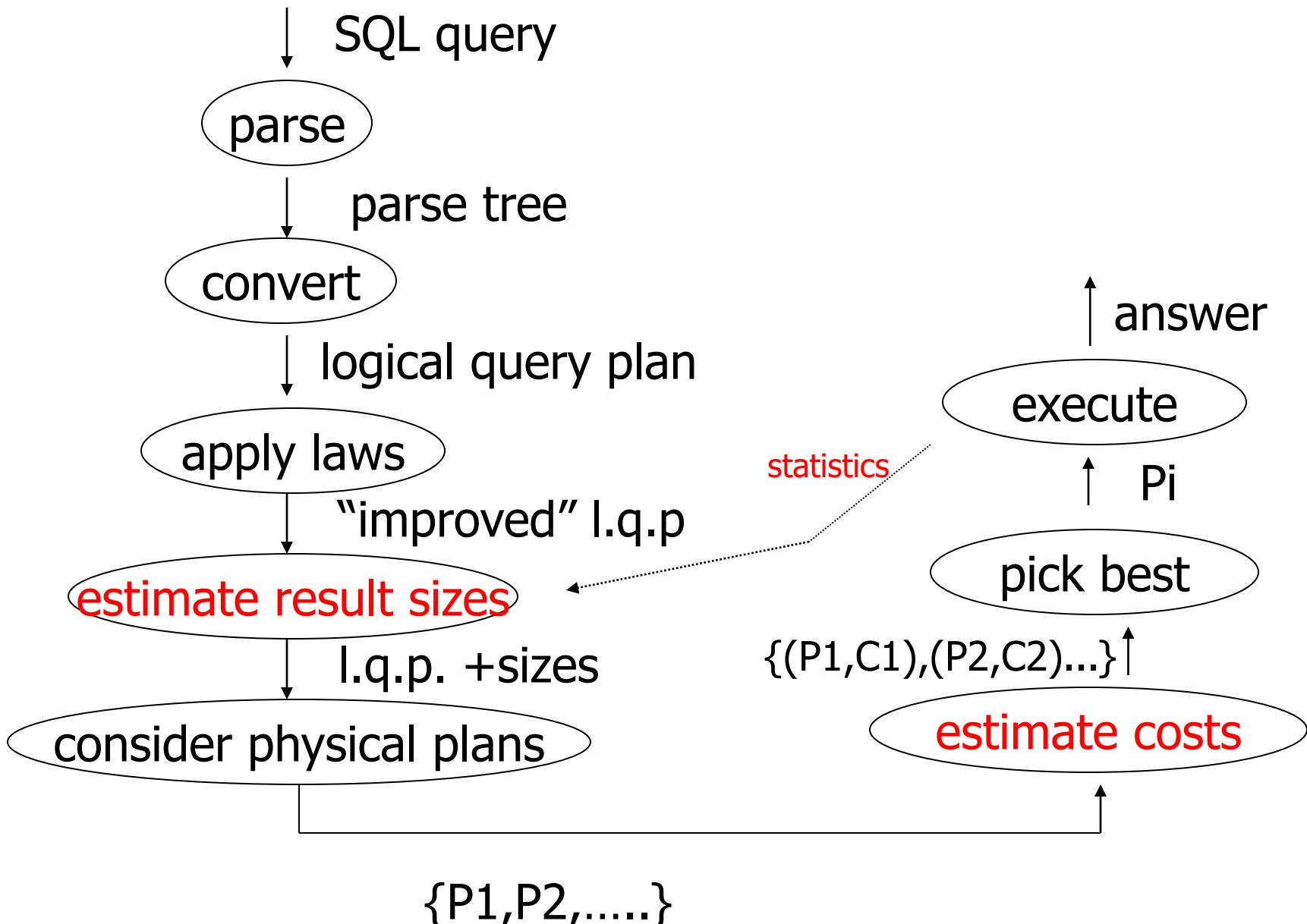
S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3





Overview of Query Optimization

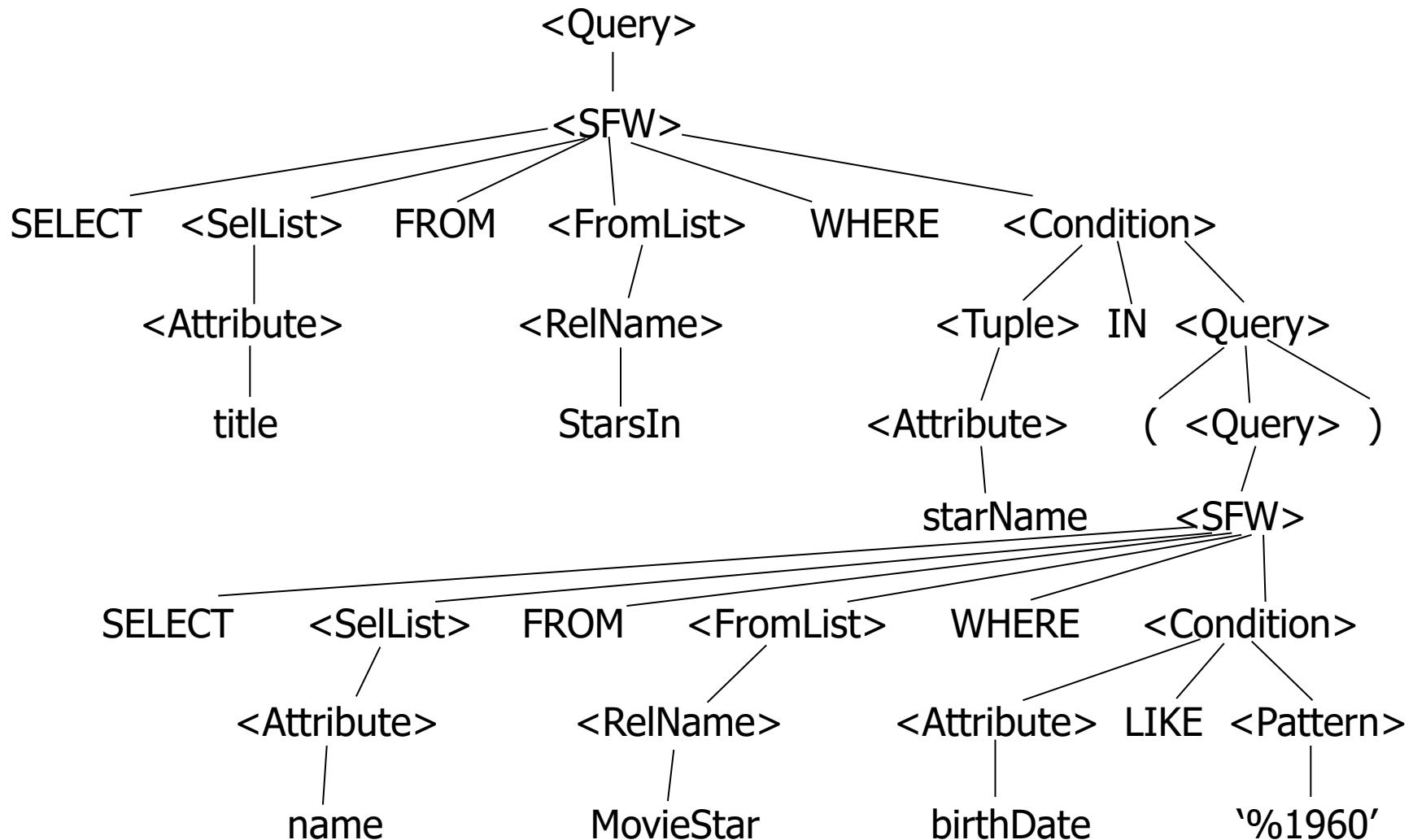


Example: SQL query

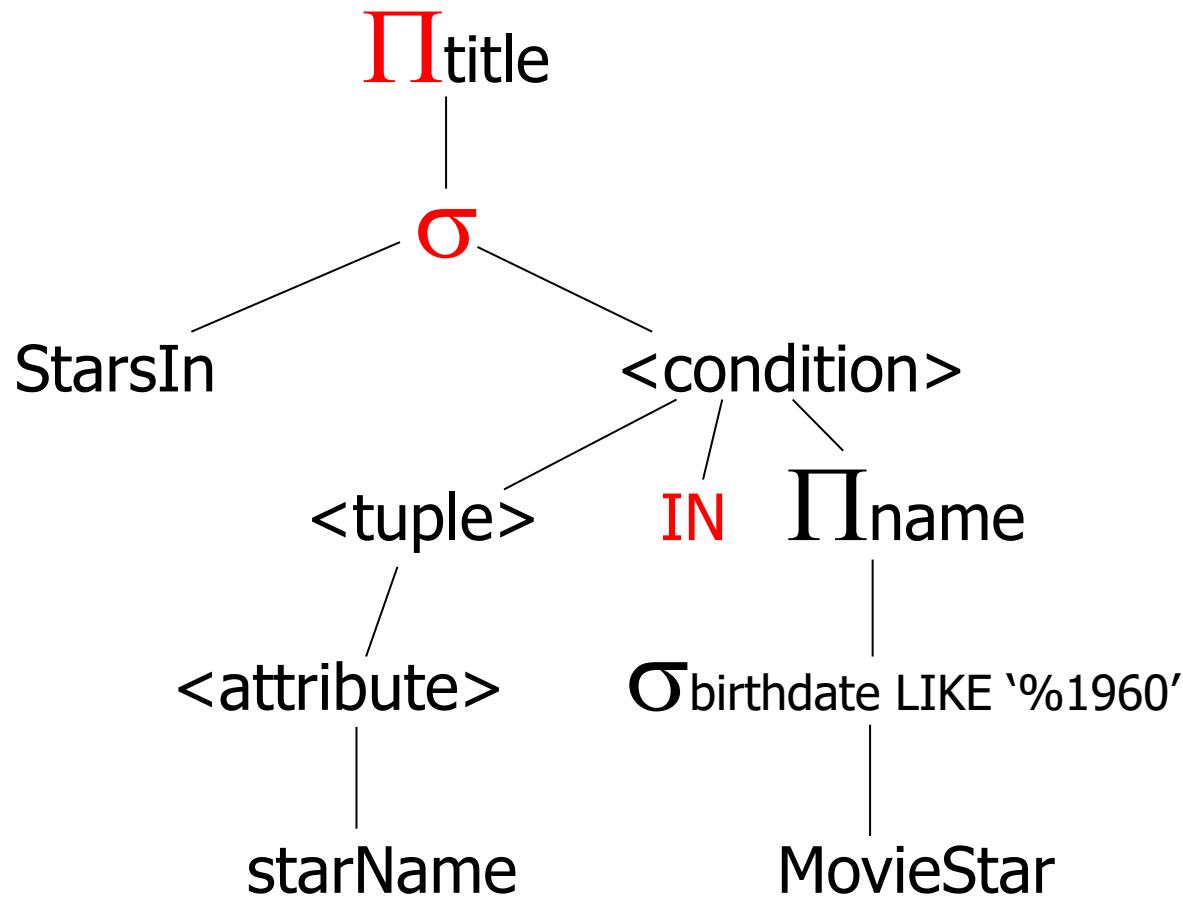
```
SELECT title
FROM StarsIn
WHERE starName IN (
    SELECT name
    FROM MovieStar
    WHERE birthdate LIKE '%1960'
);
```

(Find the movies with stars born in 1960)

Example: Parse Tree

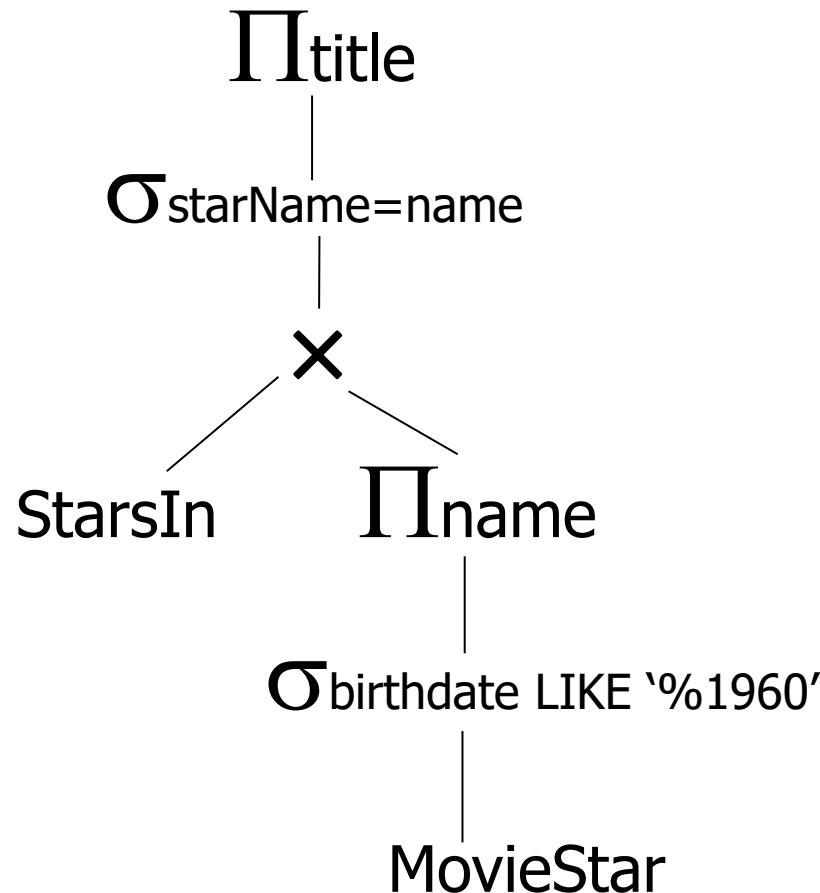


Example: Generating Relational Algebra



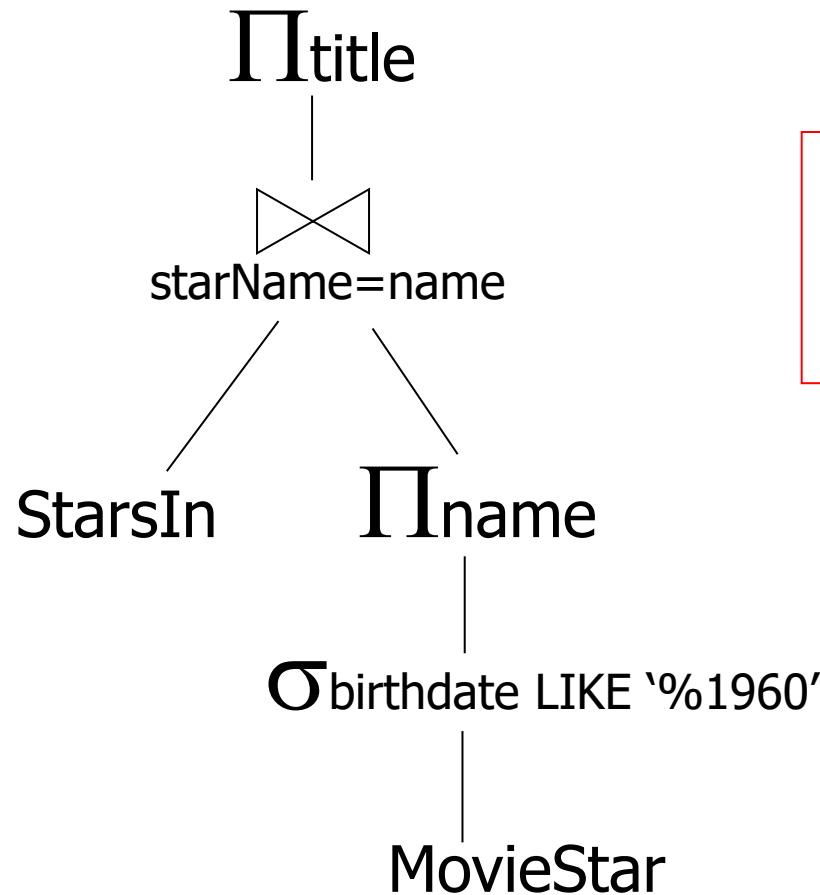
An expression using a two-argument σ , midway between a parse tree and relational algebra

Example: Logical Query Plan



Applying the rule for IN conditions

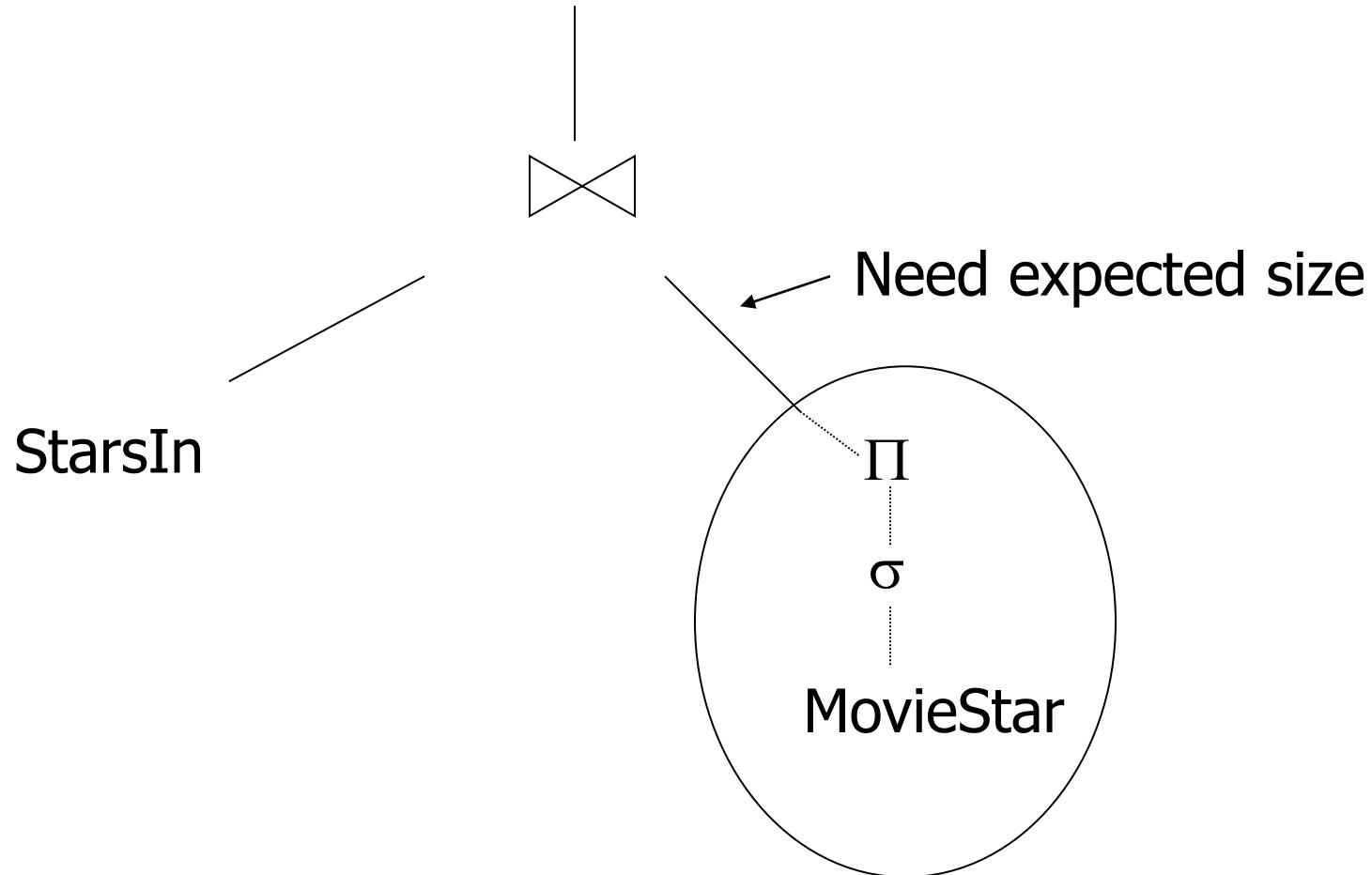
Example: Improved Logical Query Plan



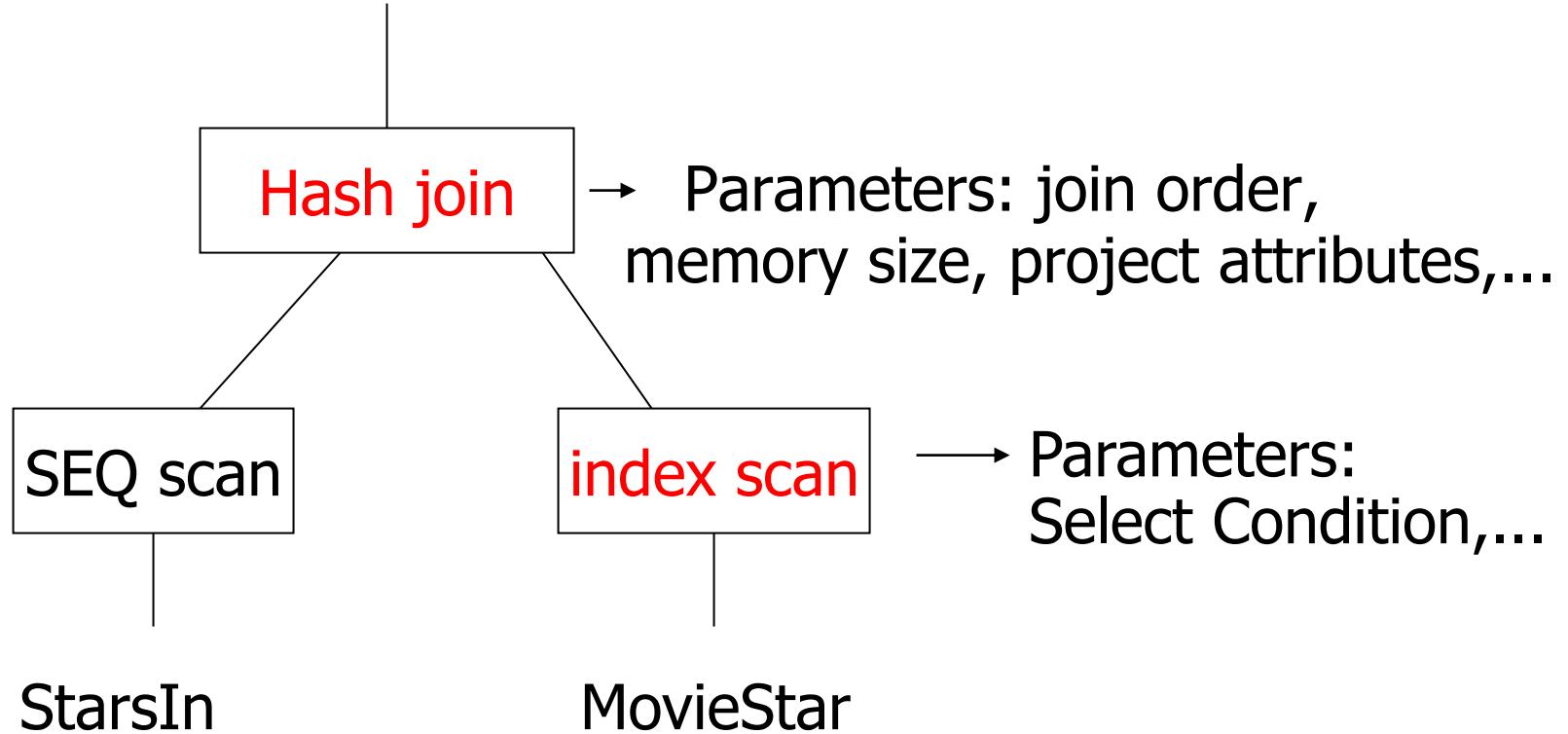
Question:
Push project to
StarsIn?

An **improvement** on previous plan

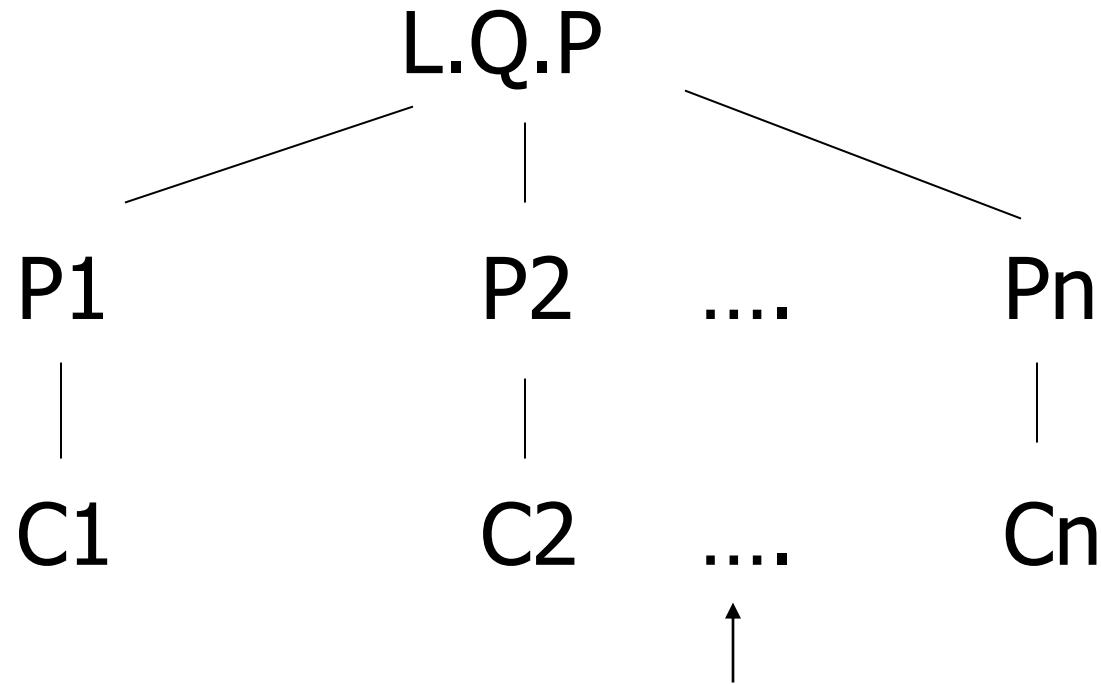
Example: Estimate Result Sizes



Example: One Physical Plan



Example: Estimate costs



Pick best!

Query Optimization

- Relational algebra level
- Detailed query plan level
 - Estimate Costs
 - without indexes
 - with indexes
 - Generate and compare plans

Relational **algebra** optimization

- Transformation rules
(preserve equivalence)
- What are good transformations?

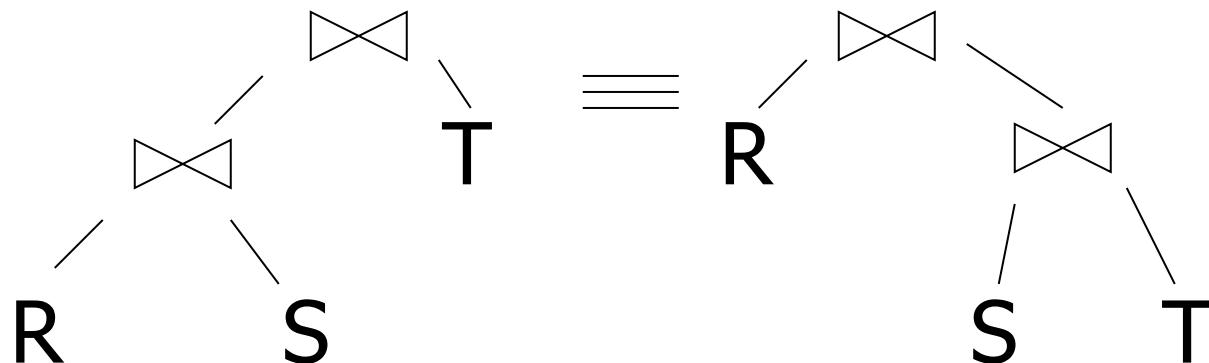
Rules: Natural joins & cross products & union

$$R \bowtie S = S \bowtie R$$

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

Note:

- Carry attribute names in results, so order is not important
- Can also write as trees, e.g.:



Rules: Natural joins & cross products & union

$$R \bowtie S = S \bowtie R$$

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

$$R \times S = S \times R$$

$$(R \times S) \times T = R \times (S \times T)$$

$$R \cup S = S \cup R$$

$$R \cup (S \cup T) = (R \cup S) \cup T$$

Rules: Selects

$$\sigma_{p_1 \wedge p_2}(R) = \sigma_{p_1} [\sigma_{p_2}(R)]$$

$$\sigma_{p_1 \vee p_2}(R) = [\sigma_{p_1}(R)] \cup [\sigma_{p_2}(R)]$$

Rules: Project

Let: $X = \text{set of attributes}$

$Y = \text{set of attributes}$

$$XY = X \cup Y$$

$$\pi_{xy}(R) = \pi_x[\pi_y(R)]$$

Rules: $\sigma + \bowtie$ combined

Let p = predicate with only R attrs

q = predicate with only S attrs

m = predicate with only R, S attrs

$$\sigma_p (R \bowtie S) = [\sigma_p (R)] \bowtie S$$

$$\sigma_q (R \bowtie S) = R \bowtie [\sigma_q (S)]$$

Rules: $\sigma + \bowtie$ combined (continued)

Some Rules can be Derived:

$$\sigma_{p \wedge q} (R \bowtie S) = (\sigma_p R) \bowtie (\sigma_q S)$$

$$\sigma_{p \wedge q \wedge m} (R \bowtie S) = \sigma_m [(\sigma_p R) \bowtie (\sigma_q S)]$$

$$\sigma_{p \vee q} (R \bowtie S) = [(\sigma_p R) \bowtie S] \cup [R \bowtie (\sigma_q S)]$$

Rules: π, σ combined

Let x = subset of R attributes

z = attributes in predicate P
(subset of R attributes)

$$\pi_x[\sigma_p(R)] = \pi_x \{ \sigma_p [\cancel{\pi_x}(R)] \}$$

Rules: π , \bowtie combined

Let x = subset of R attributes

y = subset of S attributes

z = intersection of R, S attributes

$\pi_{xy} (R \bowtie S) =$

$\pi_{xy} \{ [\pi_{xz} (R)] \bowtie [\pi_{yz} (S)] \}$

$$\pi_{xy} \{ \sigma_p (R \bowtie S) \} =$$

$$\pi_{xy} \{ \sigma_p [\pi_{xz'}(R) \bowtie \pi_{yz'}(S)] \}$$

$$z' = z \cup \{ \text{attributes used in } P \}$$

Rules for σ, π combined with X

similar...

e.g., $\sigma_p(R \times S) = ?$

Rules σ , U combined:

$$\sigma_p(R \cup S) = \sigma_p(R) \cup \sigma_p(S)$$

$$\sigma_p(R - S) = \sigma_p(R) - S = \sigma_p(R) - \sigma_p(S)$$

Which are “good” transformations?

- $\sigma_{p1 \wedge p2}(R) \rightarrow \sigma_{p1}[\sigma_{p2}(R)]$
- $\sigma_p(R \bowtie S) \rightarrow [\sigma_p(R)] \bowtie S$
- $R \bowtie S \rightarrow S \bowtie R$
- $\pi_x[\sigma_p(R)] \rightarrow \pi_x\{\sigma_p[\pi_{xz}(R)]\}$

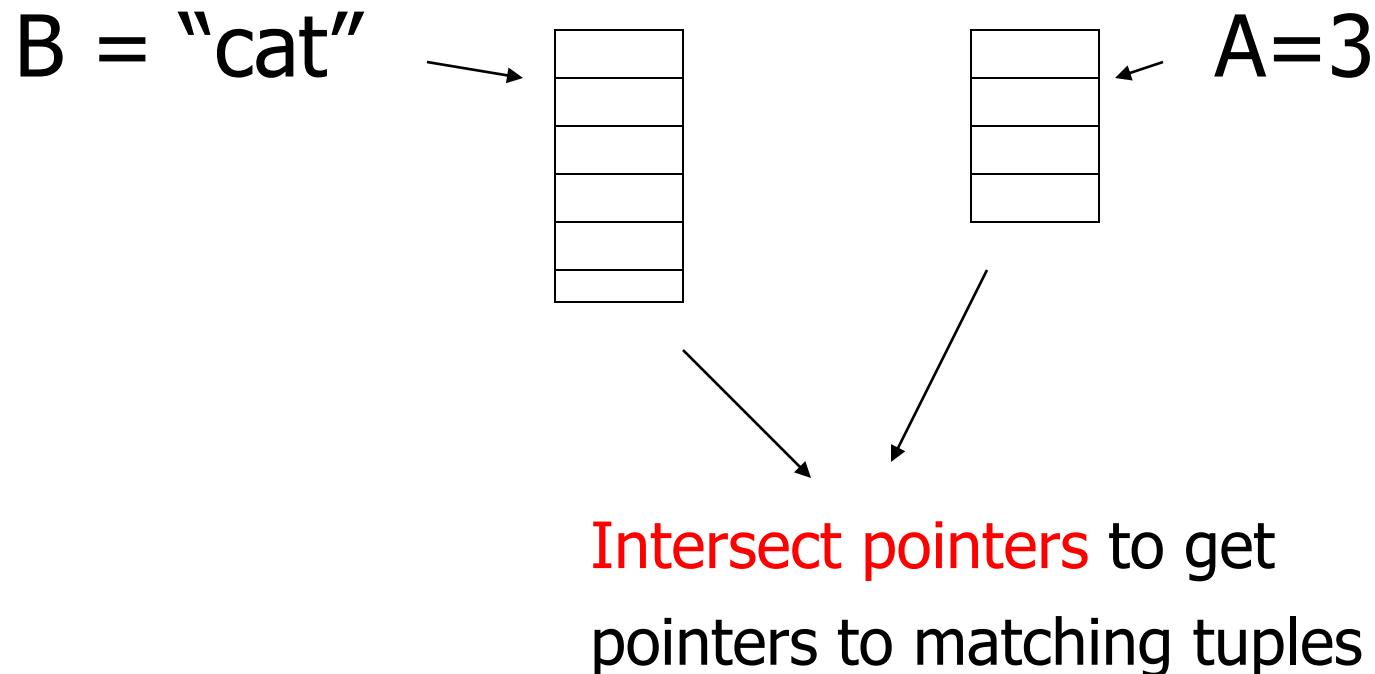
Conventional wisdom:
do **projects early**

Example: $R(A, B, C, D, E)$ $x = \{E\}$
 $P: (A=3) \wedge (B = \text{"cat"})$

$\pi_x \{\sigma_p (R)\}$ vs. $\pi_E \{\sigma_p \{\pi_{ABE}(R)\}\}$

Usually good: **early selections**

But what if we have A, B indexes?



Outline - Query Processing

- Relational algebra level
 - transformations
 - good transformations
- Detailed query plan level
 - estimate costs
 - generate and compare plans

- Estimating cost of query plan

(1) Estimating size of results

(2) Estimating # of IOs

Estimating result size

Keep statistics for relation R

- $T(R)$: # tuples in R
- $L(R)$: # of bytes in each R tuple
- $B(R)$: # of blocks to hold all R tuples
- $V(R, A)$: # distinct values in R for attribute A
- b : block size
- $bf(R)$ (blocking factor): # of tuples in a block
$$bf(R) = b/L(R)$$

Example

R

	A	B	C	D
cat	1	10	a	
cat	1	20	b	
dog	1	30	a	
dog	1	40	c	
bat	1	50	d	

A: 20 byte string

B: 4 byte integer

C: 8 byte date

D: 5 byte string

$$T(R) = 5 \quad L(R) = 37$$

$$V(R, A) = 3$$

$$V(R, C) = 5$$

$$V(R, B) = 1$$

$$V(R, D) = 4$$

Size estimates for $W = R \times S$

$$T(W) = T(R) \times T(S)$$

$$L(W) = L(R) + L(S)$$

$$bf(W) = b/(L(R)+L(S))$$

$$\begin{aligned} B(W) &= T(R)*T(S)/bf(W) = \\ &= T(R)*T(S)*L(S)/b + T(S)*T(R)*L(R)/b = \\ &= T(R)*T(S)/bf(S) + T(S)*T(R)/bf(R) = \\ &= T(R)*B(S) + T(S)*B(R) \end{aligned}$$

Size estimate for $W = \sigma_{A=a}(R)$

$$L(W) = L(R)$$

$$T(W) = ?$$

Example

R

	A	B	C	D
cat	1	10	a	
cat	1	20	b	
dog	1	30	a	
dog	1	40	c	
bat	1	50	d	

$$V(R, A) = 3$$

$$V(R, B) = 1$$

$$V(R, C) = 5$$

$$V(R, D) = 4$$

$$W = \sigma_{z=val}(R) \quad T(W) = \frac{T(R)}{V(R, Z)}$$

Selection cardinality

$SC(R,A)$ = average # records that satisfy
equality condition on $R.A$

$$SC(R,A) = T(R) / V(R,A)$$

What about $W = \sigma_{z \geq \text{val}}(R)$?

$$T(W) = ?$$

- Solution # 1: (not bad)
 $T(W) = T(R)/2$
- Solution # 2: (better)
 $T(W) = T(R)/3$

- Solution # 3: Estimate values in range

Example R

	Z

$$\text{Min}=1 \quad V(R, Z)=10$$



$$\text{Max}=20$$

$$W = \sigma_{z \geq 15} (R)$$

$$f = \frac{20-15+1}{20-1+1} = \frac{6}{20} \quad (\text{fraction of range})$$

$$T(W) = f \times T(R) \quad (\text{best, if we know } \text{Dom}(Z))$$

Equivalently:

$f \times V(R, Z)$ = fraction of distinct values

$$T(W) = [f \times V(Z, R)] \times \frac{T(R)}{V(Z, R)} = f \times T(R)$$

Size estimate for $W = R1 \bowtie R2$

Let x = attributes of $R1$

y = attributes of $R2$

Case 1

$$X \cap Y = \emptyset$$

Same as $R1 \times R2$

Case 2

$$W = R1 \bowtie R2 \quad X \cap Y = A$$

R1	A	B	C

R2	A	D

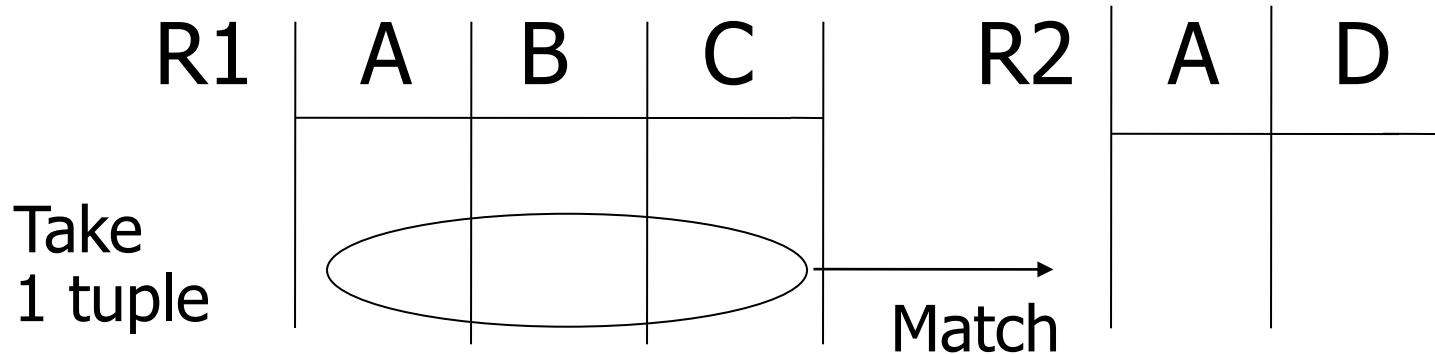
Assumption:

$V(R1, A) \leq V(R2, A) \Rightarrow$ Every A value in R1 is in R2

$V(R2, A) \leq V(R1, A) \Rightarrow$ Every A value in R2 is in R1

“containment of value sets”

Computing $T(W)$ when $V(R1, A) \leq V(R2, A)$



1 tuple matches with $\frac{T(R2)}{V(R2, A)}$ tuples...

so $T(W) = \frac{T(R2)}{V(R2, A)} \times T(R1)$

- $V(R1, A) \leq V(R2, A) \quad T(W) = \frac{T(R2) T(R1)}{V(R2, A)}$
- $V(R2, A) \leq V(R1, A) \quad T(W) = \frac{T(R2) T(R1)}{V(R1, A)}$

[A is common attribute]

In general $W = R1 \bowtie R2$

$$T(W) = \frac{T(R2) \ T(R1)}{\max\{ V(R1,A), V(R2,A) \}}$$

Size Estimation Summary (1/2)

$$\sigma_{A=v}(R) \quad \text{SC(R,A)} \quad (\rightarrow \text{SC(R,A)} = T(R) / V(R,A))$$

$$\sigma_{A \leq v}(R) \quad T(R) * \frac{v - \min(A, R)}{\max(A, R) - \min(A, R)}$$

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(R) \quad \text{multiplying probabilities}$$

$$T(R) * [(sc_1/T(R)) * (sc_2/T(R)) * \dots * (sc_n/T(R))]$$

$$\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(R) \quad \text{probability that a record satisfy none of } \theta:$$

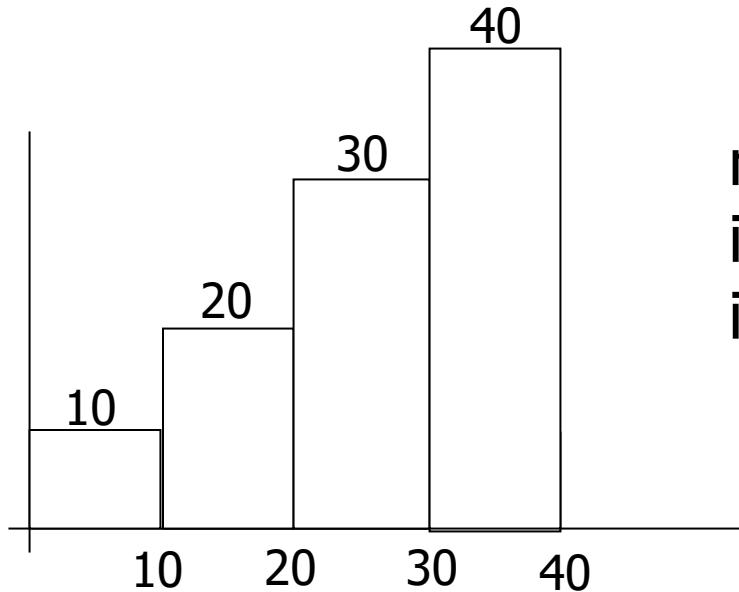
$$[(1 - sc_1/T(R)) * (1 - sc_2/T(R)) * \dots * (1 - sc_n/T(R))]$$

$$T(R) * (1 - [(1 - sc_1/T(R)) * (1 - sc_2/T(R)) * \dots * (1 - sc_n/T(R))])$$

Size Estimation Summary(2/2)

- $R \times S$
 $T(R \times S) = T(R) * T(S)$
- $R \bowtie S$
 - $R \cap S = \emptyset$: $T(R) * T(S)$
 - $R \cap S$ key for R : maximum output size is $T(S)$
 - $R \cap S$ foreign key for R : $T(S)$
 - $R \cap S = \{A\}$, neither key of R nor S
 - $T(R) * T(S) / V(S, A)$
 - $T(R) * T(S) / V(R, A)$

A Note on **Histograms**



number of tuples
in R with A value
in given range

$$\sigma_{A=val}(R) = ?$$

Summary

- Estimating size of results is an “art”
- Don’t forget:
Statistics must be kept up to date...
(cost?)

Execution

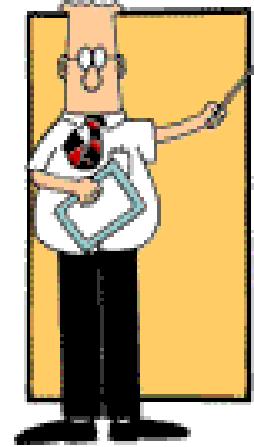
Oracle SQL ~~Tuning~~

An Introduction



Overview

- **Foundation**
 - Optimizer, cost vs. rule, data storage, SQL-execution phases, ...
- **Creating & reading execution plans**
 - Access paths, single table, joins, ...
- **Utilities**
 - Tracefiles, SQL hints, analyze/dbms_stat
- **Warehouse specifics**
 - Star queries & bitmap indexing
 - ETL
- **Availability in version 9, 10, 11, 12 ?**



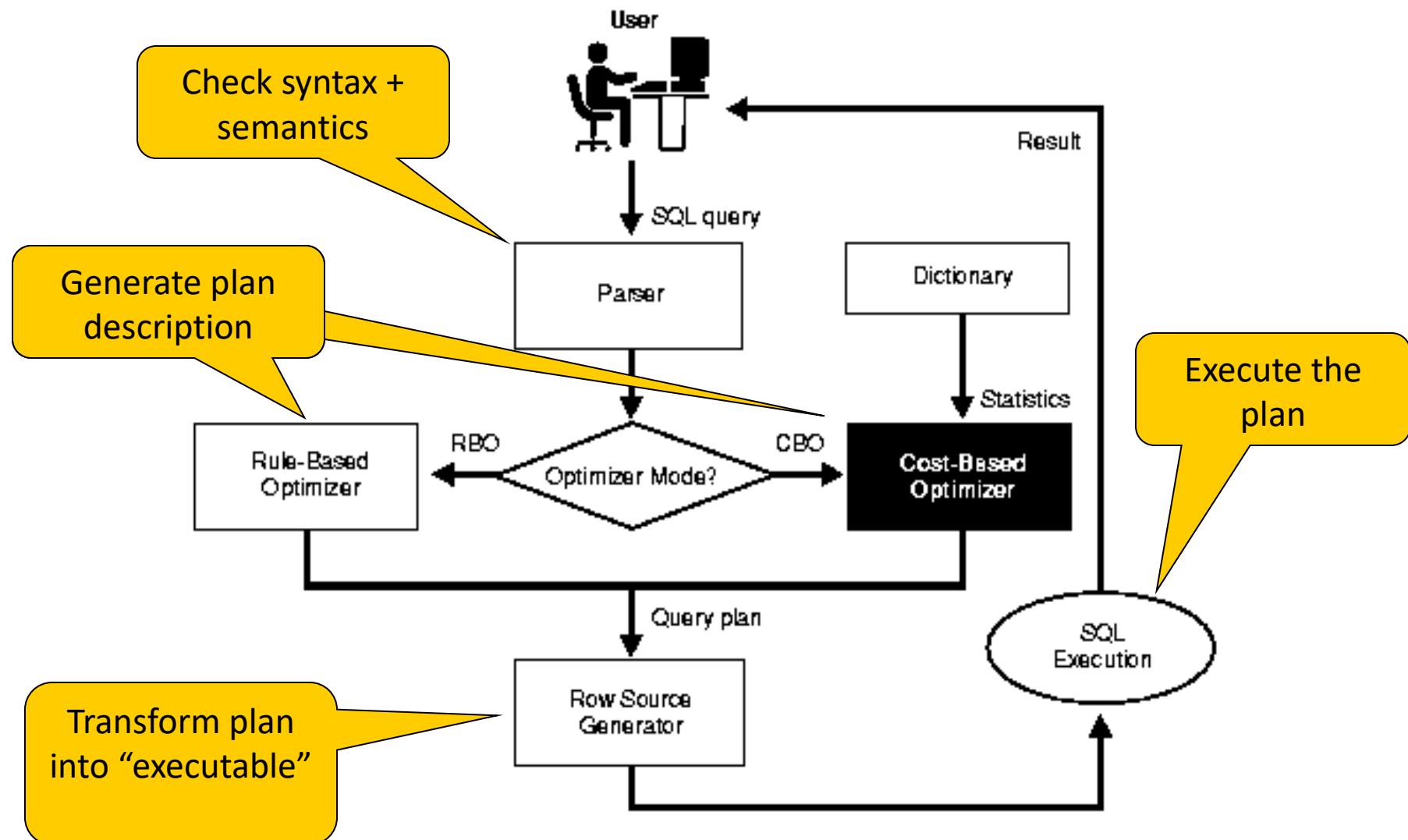
Goals

- Read execution plans
 - Table access
 - Index access
 - Joins
 - Subqueries
- Understand execution plans
 - Understand performance
 - Basic understanding of SQL optimization
- Start thinking how you should have executed it

Next...

- Basic Concepts (13)
 - Background information
- SQL-Execution (50)
 - Read + understand

Optimizer Overview



Cost vs. Rule

- Rule (RBO: Rule Based Optimization)
 - Hardcoded heuristic rules determine plan
 - “Access via **index is better than full table scan**”
 - “Fully matched index is better than partially matched index”
 - ...
- Cost (2 modes)
 - **Statistics of data** play role in plan determination
 - Best throughput mode: retrieve **all rows** asap
 - First compute, then return fast
 - Best response mode: retrieve **first row** asap
 - Start returning while computing (if possible)

How to set which one?

- Instance level: Optimizer_Mode parameter
 - Rule
 - Choose
 - if statistics then CBO (all_rows), else RBO
 - First_rows, First_rows_n (1, 10, 100, 1000)
 - All_rows
- Session level:
 - Alter session set optimizer_mode=<mode>;
- Statement level:
 - **Hints inside SQL** text specify mode to be used

DML vs. Queries

- Open => Parse => Execute (=> Fetchⁿ)

```
SELECT ename, salary  
FROM emp  
WHERE salary>100000
```

Fetches done
By client

```
UPDATE emp  
SET commission='N'  
WHERE salary>100000
```

Same SQL optimization

All fetches done internally
by SQL-Executor

CLIENT

=> SQL =>
<= Data or Returncode<=

SERVER

Data Storage: Tables

- Oracle stores all data inside datafiles
 - Location & size determined by DBA
 - Logically grouped in **tablespaces**
 - Each file is identified by a relative **file number** (fno)
- Datafile consists of data-blocks
 - Size equals value of *db_block_size* parameter
 - Each **block** is identified by its **offset in the file**
- Data-blocks contain rows
 - Each **row** is identified by its **sequence in the block**

ROWID: <Block>.<Row>.<File>

Data Storage: Tables

File x

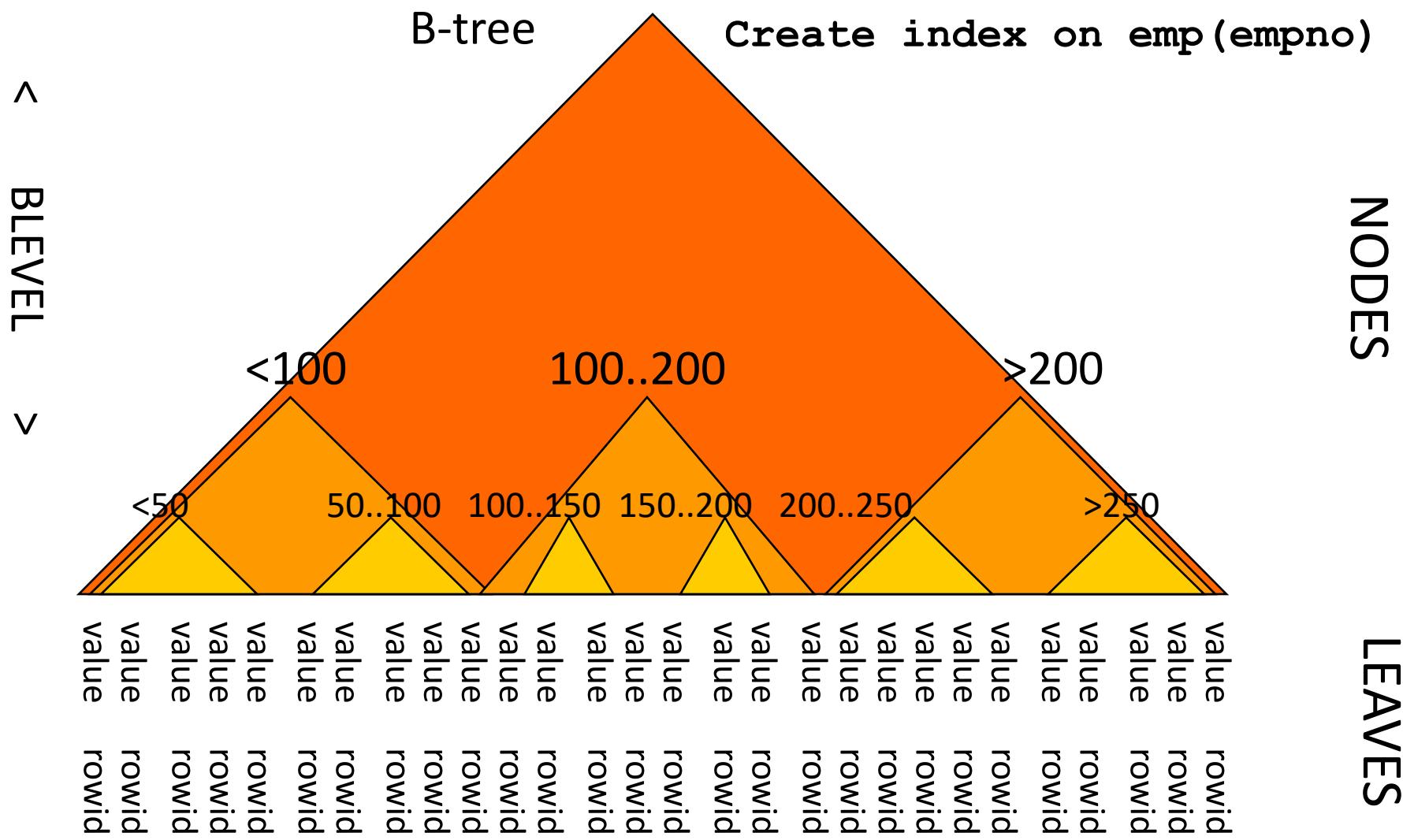
Block 1	Block 2	Block 3	Block 4
Block 5	Block 6	<p><Rec1><Rec2><Rec3></p> <p><Rec4><Rec5><Rec6></p> <p><Rec7><Rec8><Rec9></p> <p>...</p>	

Rowid: 00000007.0000.000X

Data Storage: Indexes

- **Balanced trees**
 - Indexed column(s) sorted and stored separately
 - **NULL values are excluded** (not added to the index)
 - Pointer structure enables logarithmic search
 - Access index first, **find pointer** to table, **then access table**
- B-trees consist of
 - Node blocks
 - Contain pointers to other node, or leaf blocks
 - **Leaf blocks**
 - Contain actual **indexed values**
 - Contain **rowids** (pointer to rows)
- Also stored in blocks in datafiles
 - Proprietary format

Data Storage: Indexes



Data Storage: Indexes

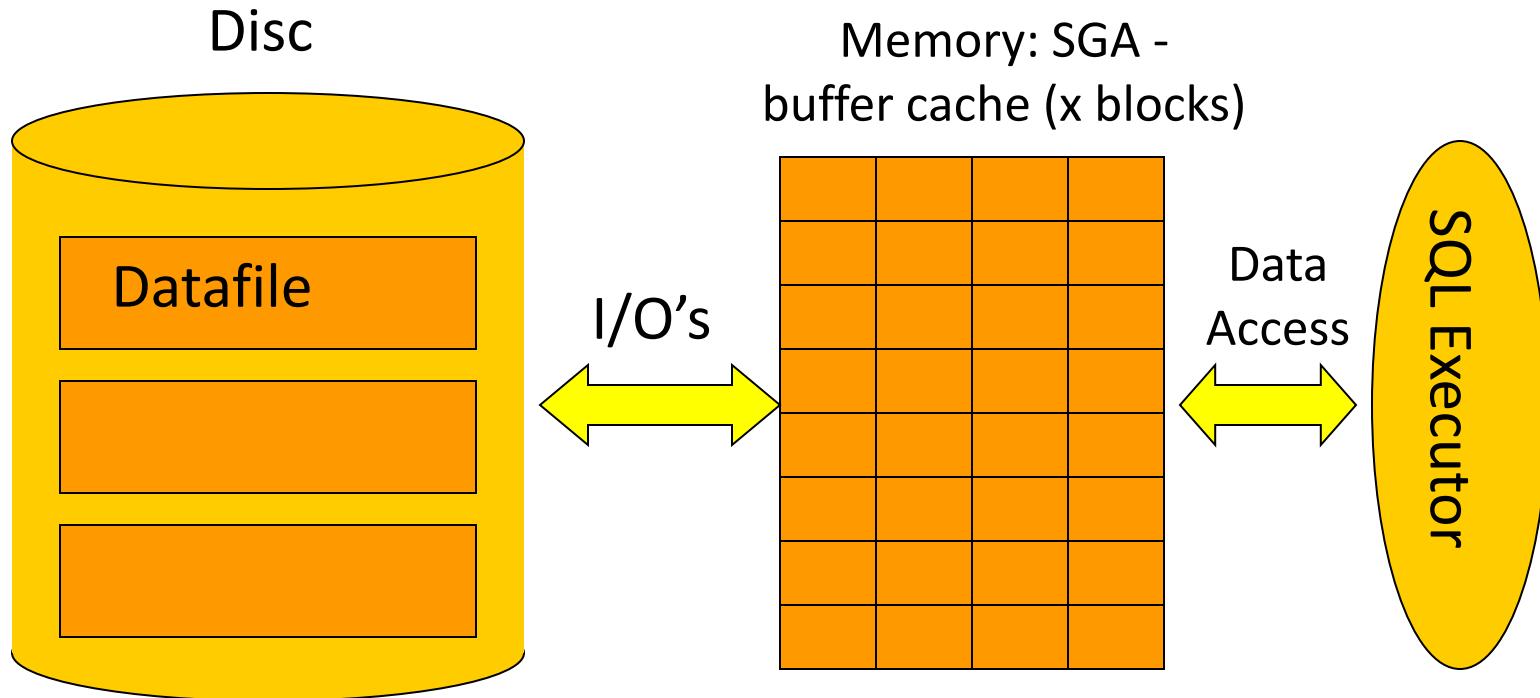
Datafile

Block 1	Block 2	Block 3	Block 4
Block 5	Block ...	Index Node Block	Index Leaf Block
Index Leaf Block			

No particular order of node and leaf blocks

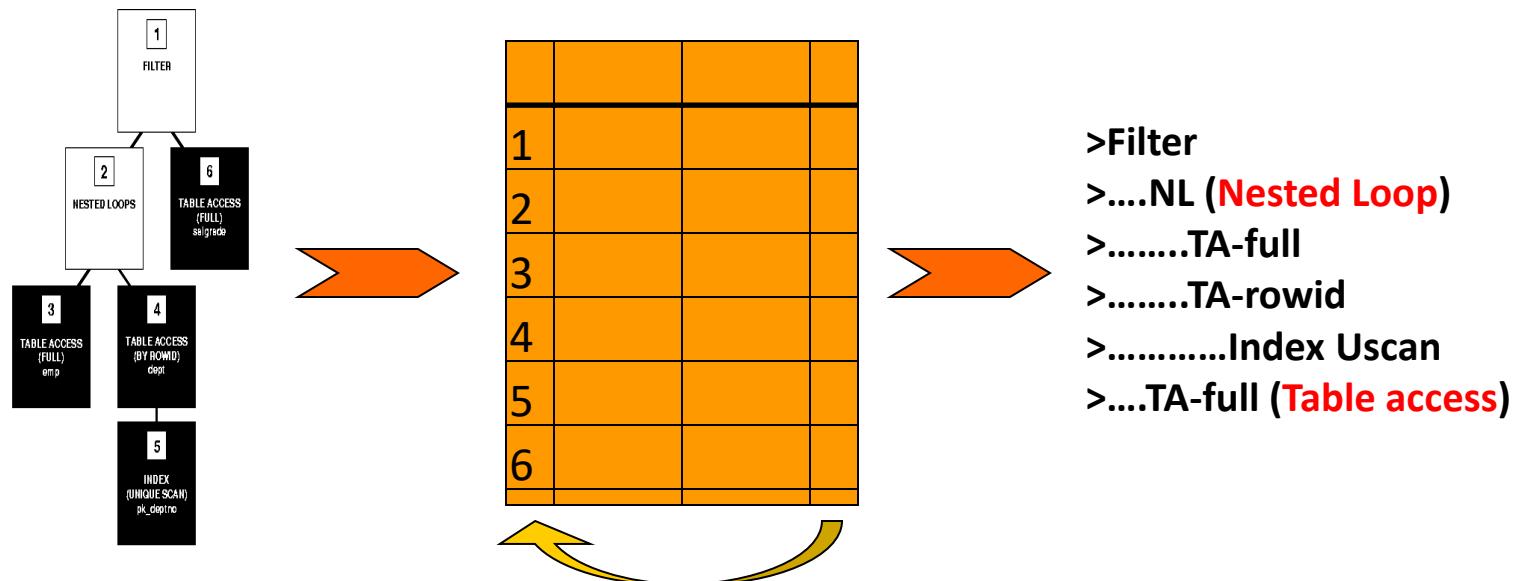
Table & Index I/O

- I/O's are done at 'block level'
 - LRU list controls who 'makes place' in the cache



Explain Plan Utility

- “Explain plan for <SQL-statement>”
 - Stores plan (row-sources + operations) in Plan_Table
 - View on Plan_Table (or 3rd party tool) formats into readable plan



Explain Plan Utility

```
create table PLAN_TABLE (
    statement_id      varchar2(30),      operation      varchar2(30),
    options          varchar2(30),      object_owner  varchar2(30),
    object_name      varchar2(30),      id            numeric,
    parent_id        numeric,          position      numeric,
    cost             numeric,          bytes         numeric);
```

```
create or replace view PLANS(STATEMENT_ID,PLAN,POSITION) as
select statement_id,
       rpad('>',2*level,'.')||operation||
       decode(options,NULL,' ',' (')||nvl(options,' ')|||
       decode(options,NULL,' ',''))|||
       decode(object_owner,NULL,'',object_owner||'.')||object_name plan,
       position
  from plan_table
 start with id=0
 connect by prior id=parent_id
        and prior nvl(statement_id,'NULL')=nvl(statement_id,'NULL')
```

Execution Plans

1. Single table without index
2. Single table with index
3. Joins
 1. Nested Loop
 2. Sort Merge
 3. Hash1 (small/large), hash2 (large/large)
4. Special operators

Single Table, no Index (1.1)

```
SELECT *  
FROM emp;
```

```
>.SELECT STATEMENT  
>...TABLE ACCESS full emp
```

- Full table scan (**FTS**)
 - All blocks read sequentially into buffer cache
 - Also called “buffer-gets”
 - Done via multi-block I/O’s (db_file_multiblock_read_count)
 - Till high-water-mark reached (truncate resets, delete not)
 - Per block: extract + return all rows
 - Then put block at **LRU-end** of LRU list (!)
 - **All other operations** put block at **MRU-end**

Single Table, no Index (1.2)

```
SELECT *  
FROM emp  
WHERE sal > 100000;
```

```
>.SELECT STATEMENT  
>...TABLE ACCESS full emp
```

- Full table scan with filtering
 - Read all blocks
 - Per block extract, filter, then return row
 - **Simple where-clause filters never shown in plan**
 - FTS with: **rows-in > rows-out**

Single Table, no Index (1.3)

```
SELECT *  
FROM emp  
ORDER BY ename;
```

```
>.SELECT STATEMENT  
>...SORT order by  
>....TABLE ACCESS full emp
```

- FTS followed by sort on ordered-by column(s)
 - “Followed by” ie. **SORT won’t return rows to its parent** row-source **till** its child row-source **fully completed**
 - SORT order by: rows-in = rows-out
 - Small sorts done in memory (SORT_AREA_SIZE)
 - Large sorts done via **TEMPORARY** tablespace
 - Potentially many I/O’s

Single Table, no Index (1.3)

```
SELECT *
FROM emp
ORDER BY ename;
```

Emp(ename)

```
>.SELECT STATEMENT
>...TABLE ACCESS full emp
>....INDEX full scan i_emp_ename
```

- If ordered-by column(s) is indexed
 - Index Full Scan
 - CBO uses index if mode = **First_Rows**
 - If index is used => **no sort** is necessary

Single Table, no Index (1.4)

```
SELECT job,sum(sal)  
FROM emp  
GROUP BY job;
```

```
>. SELECT STATEMENT  
>... SORT group by  
>.... TABLE ACCESS full emp
```

- FTS followed by sort on grouped-by column(s)
 - FTS will only retrieve job and sal columns
 - Small intermediate rowlength => sort more likely in memory
 - SORT group by: **rows-in >> rows-out**
 - Sort also computes aggregates

Single Table, no Index (1.5)

```
SELECT job,sum(sal)  
FROM emp  
GROUP BY job  
HAVING sum(sal)>200000;
```

```
>.SELECT STATEMENT  
>...FILTER  
>.....SORT group by  
>.....TABLE ACCESS full emp
```

- HAVING Filtering
 - Only **filter rows** that comply to having-clause

Single Table, no Index (1.6)

```
SELECT *
FROM emp
WHERE rowid=
  '00004F2A.00A2.000C'
```

```
>. SELECT STATEMENT
>... TABLE ACCESS by rowid emp
```

- Table access by rowid
 - Single row lookup
 - **Goes straight to the block**, and filters the row
 - Fastest way to retrieve one row
 - If you know its rowid

Single Table, Index (2.1)

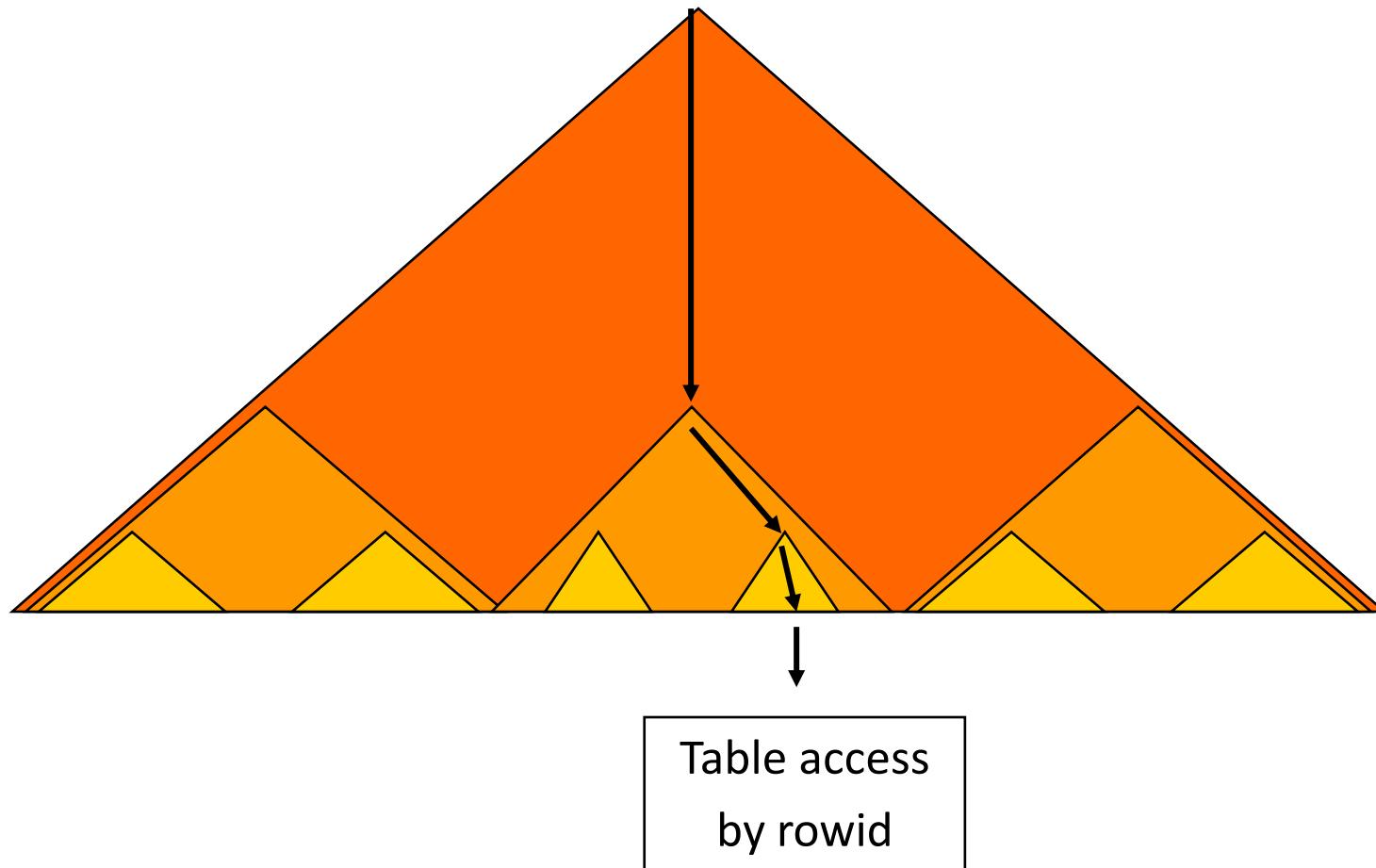
```
SELECT *
FROM emp
WHERE empno=174;
```

Unique emp(empno)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>....INDEX unique scan i_emp_pk
```

- Index **Unique** Scan
 - Traverses the node blocks to locate correct leaf block
 - Searches value in leaf block (if not found => done)
 - Returns rowid to parent row-source
 - Parent: accesses the file+block and returns the row

Index Unique Scan (2.1)

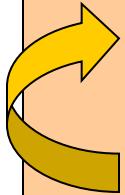


Single Table, Index (2.2)

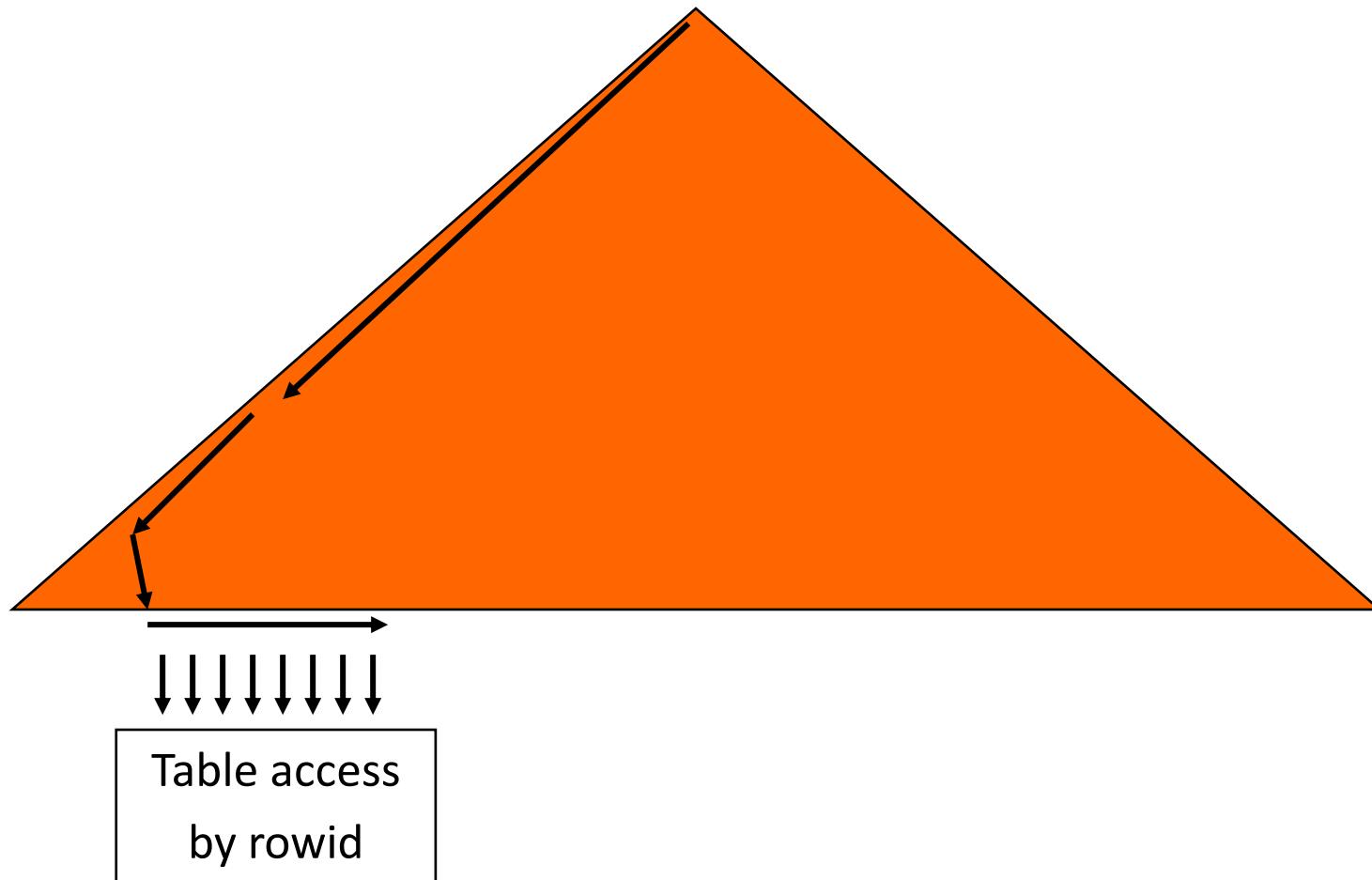
```
SELECT *
FROM emp
WHERE job='manager';
```

emp(job)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>....INDEX range scan i_emp_job
```

- **(Non-unique) Index Range Scan**
 - Traverses the node blocks to locate most left leaf block
 - Searches 1st occurrence of value in leaf block
 - Returns rowid to parent row-source
 - Parent: accesses the file+block and returns the row
 - **Continues on to next occurrence of value in leaf block**
 - Until no more occurrences
- 

Index Range Scan (2.2)



Single Table, Index (2.3)

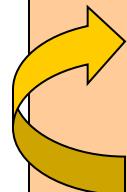
```
SELECT *
FROM emp
WHERE empno>100;
```

Unique emp(empno)

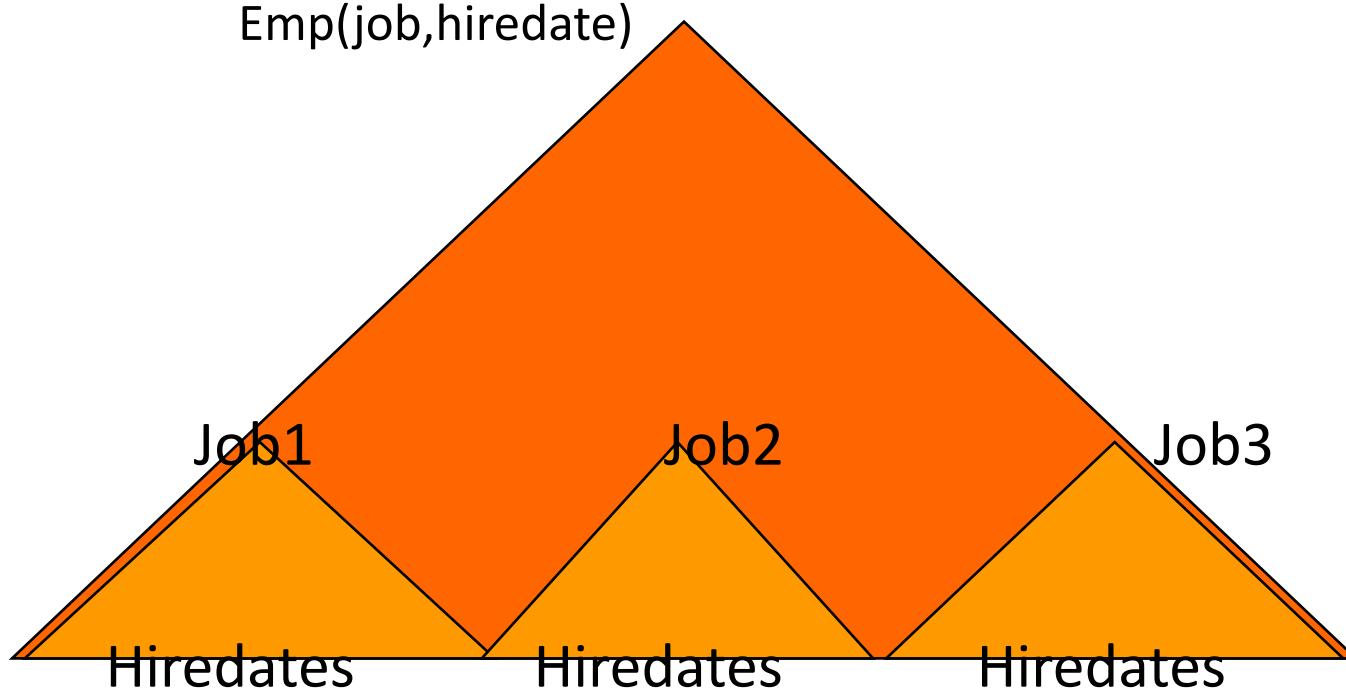
```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>....INDEX range scan i_emp_pk
```

- Unique Index Range Scan

- Traverses the node blocks to **locate most left** leaf block with start value
- Searches 1st occurrence of value-range in leaf block
- Returns rowid to parent row-source
 - Parent: accesses the file+block and returns the row
- Continues on to next valid occurrence in leaf block
 - Until no more occurrences / no longer in value-range



Concatenated Indexes



Multiple levels of Btrees, by column order

Single Table, Index (2.4)

```
SELECT *
FROM emp
WHERE job='manager'
AND hiredate='01-01-2001';
```

Emp(job,hiredate)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>....INDEX range scan i_emp_j_h
```

- Full **Concatenated Index**
 - Use job-value to navigate to sub-Btree
 - Then search all applicable hiredates

Single Table, Index (2.5)

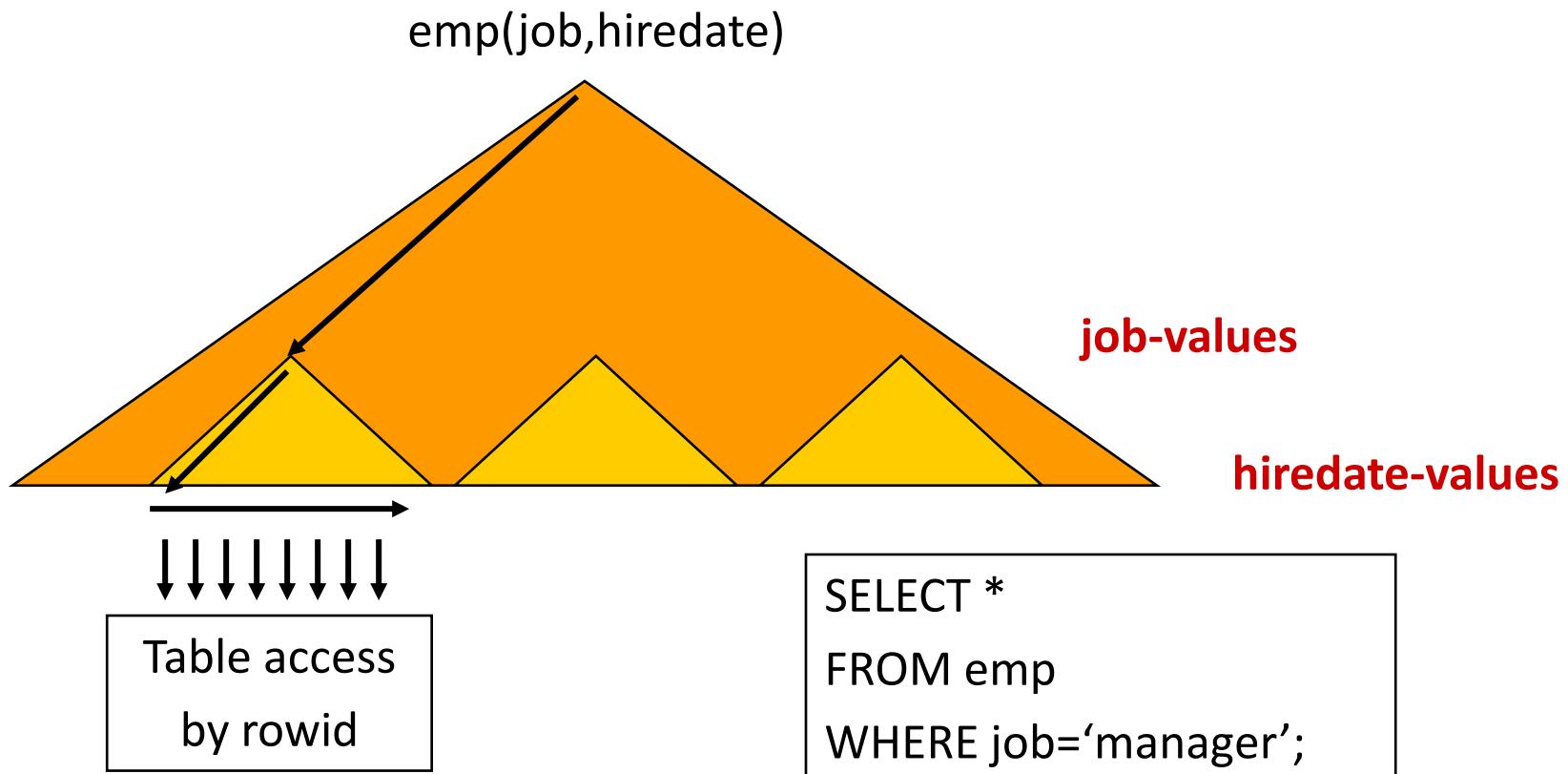
```
SELECT *
FROM emp
WHERE job='manager';
```

Emp(job,hiredate)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>....INDEX range scan i_emp_j_h
```

- (Leading) **Prefix** of Concatenated Index
 - Scans **full sub-Btree** inside larger Btree

Index Range Scan (2.5)



Single Table, Index (2.6)

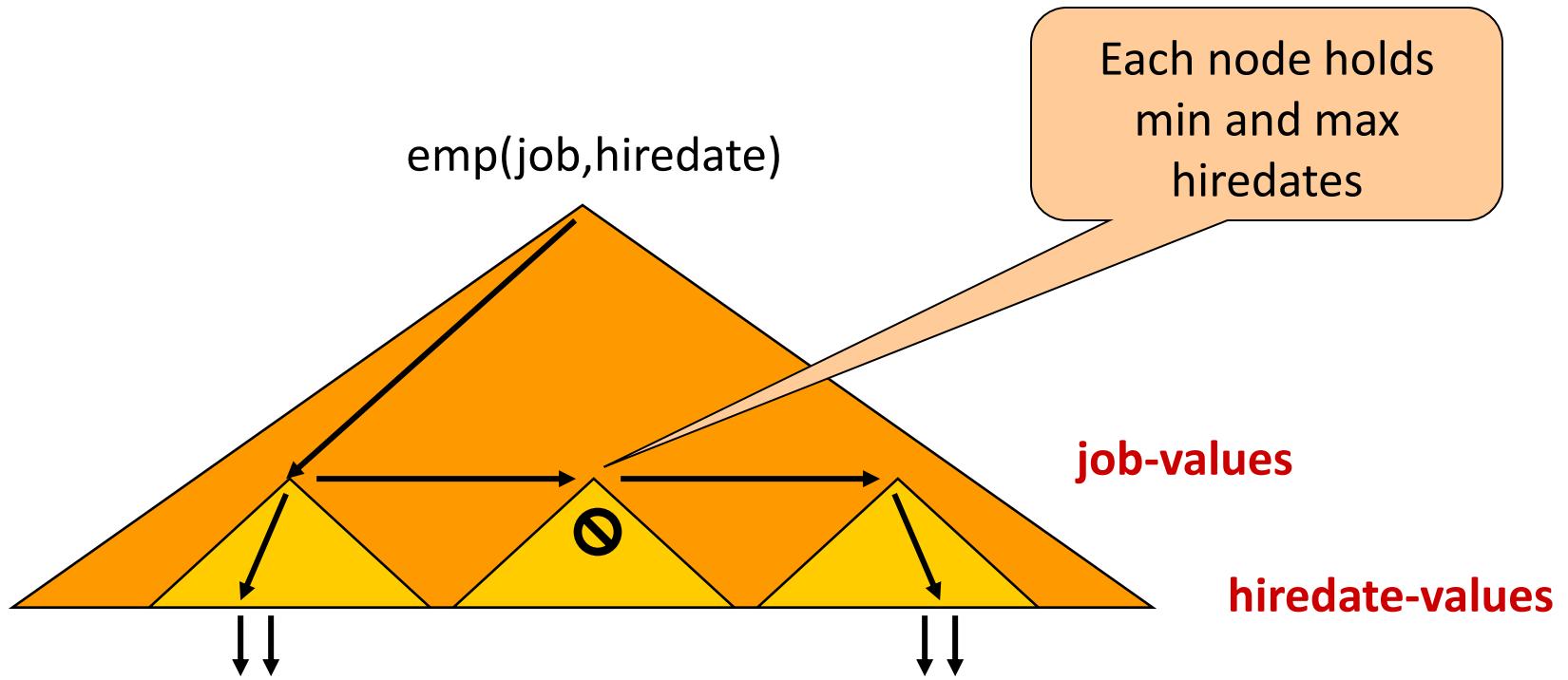
```
SELECT *
  FROM emp
 WHERE hiredate='01-01-2001';
```

Emp(job,hiredate)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_j_h
```

- Index **Skip Scan** (prior versions did FTS)
 - “To use indexes where they’ve never been used before”
 - Predicate on leading column(s) no longer needed
 - Views Btree as collection of smaller sub-Btrees
 - Works best with low-cardinality leading column(s)

Index Skip Scan (2.6)



```
SELECT *
FROM emp
WHERE hiredate='01-01-2001';
```

Single Table, Index (2.7)

```
SELECT *
FROM emp
WHERE empno>100
AND job='manager';
```

Unique Emp(empno)
Emp(job)

```
>.SELECT STATEMENT
>... TABLE ACCESS by rowid emp
>.... INDEX range scan i_emp_job
```

- Multiple Indexes
 - Rule: uses heuristic decision list to choose which one
 - Available indexes are ‘ranked’
 - Cost: computes **most selective one** (ie. least costing)
 - **Uses statistics**

RBO Heuristics

- Ranking multiple available indexes
 1. Equality on single column unique index
 2. Equality on concatenated unique index
 3. Equality on concatenated index
 4. Equality on single column index
 5. Bounded range search in index
 - Like, Between, Leading-part, ...
 6. Unbounded range search in index
 - Greater, Smaller (on leading part)

Normally you hint which one to use

CBO Cost Computation

- Statistics at various levels
 - Table:
 - Num_rows, Blocks, Empty_blocks, Avg_space
 - Column:
 - Num_values, Low_value, High_value, Num_nulls
 - Index:
 - Distinct_keys, Blevel, Avg_leaf_blocks_per_key, Avg_data_blocks_per_key, Leaf_blocks
 - Used to compute selectivity of each index
 - **Selectivity = percentage of rows returned**
 - Number of I/O's plays big role
 - FTS is also considered at this time!

Single Table, Index (2.1)

```
SELECT *
FROM emp
WHERE empno=174;
```

Unique emp(empno)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>....INDEX unique scan i_emp_pk
Or,
>.SELECT STATEMENT
>...TABLE ACCESS full emp
```

- CBO will use Full Table Scan If,
of I/O's to do FTS < # of I/O's to do IRS (Index Range Scan)
 - FTS I/O uses db_file_multiblock_read_count (dfmrc)
 - Typically 16
 - Unique scan uses: $(\text{blevel} + 1) + 1$ I/O's
 - FTS uses $\text{ceil}(\#\text{table blocks} / \text{dfmrc})$ I/O's

CBO: Clustering Factor

- Index level statistic

- How well ordered are the rows in comparison to indexed values?
- Average number of blocks to access a single value
 - 1 means range scans are cheap
 - <# of table blocks> means range scans are expensive
- Used to rank multiple available range scans

Blck 1	Blck 2	Blck 3
-----	-----	-----
A A A	B B B	C C C

Clust.factor = 1

Blck 1	Blck 2	Blck 3
-----	-----	-----
A B C	A B C	A B C

Clust.factor = 3

Single Table, Index (2.2)

```
SELECT *
FROM emp
WHERE job='manager';
```

emp(job)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>....INDEX range scan i_emp_job
Or,
>.SELECT STATEMENT
>...TABLE ACCESS full emp
```

- Clustering factor comparing IRS against FTS
 - If, (#table blocks / dfmrc) $<$
$$(\#values * clust.factor) + blevel + leafblocks-to-visit$$
 then, FTS is used

Single Table, Index (2.7)

```
SELECT *
FROM emp
WHERE empno>100
AND job='manager';
```

Unique Emp(empno)
Emp(job)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>....INDEX range scan i_emp_job
Or,
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>....INDEX range scan i_emp_empno
```

- Clust.factor comparing multiple IRS's
 - Suppose FTS is too many I/O's
 - Compare (#values * clust.fact) **to decide which index**
 - Empno-selectivity => #values * 1 => # I/O's
 - Job-selectivity => 1 * clust.fact => # I/O's

Single Table, Index (2.8)

```
SELECT *
FROM emp
WHERE job='manager'
AND depno=10
```

Emp(job)
Emp(depno)

```
>.SELECT STATEMENT
>...TABLE ACCESS by rowid emp
>....AND-EQUAL
>.....INDEX range scan i_emp_job
>.....INDEX range scan i_emp_depno
```

- Multiple same-rank, single-column indexes
 - **AND-EQUAL**: merge up to 5 single column range scans
 - Combines multiple index range scans prior to table access
 - Intersects rowid sets from each range scan
 - Rarely seen with CBO

Single Table, Index (2.9)

```
SELECT ename  
FROM emp  
WHERE job='manager';
```

Emp(job,ename)

```
>.SELECT STATEMENT  
>...INDEX range scan i_emp_j_e
```

- Using indexes to avoid table access
 - Depending on columns used in SELECT-list and other places of WHERE-clause
 - No table-access if all used columns present in index

Single Table, Index (2.10)

```
SELECT count(*)  
FROM big_emp;
```

Big_emp(empno)

```
>.SELECT STATEMENT  
>...INDEX fast full scan i_emp_empno
```

- Fast Full Index Scan (CBO only)
 - Uses same multiblock I/O as FTS
 - Eligible index must have **at least one NOT NULL column**
 - Rows are returned **leaf-block order**
 - Not in indexed-columns-order

Joins, Nested Loops (3.1)

```
SELECT *  
FROM dept, emp;
```

```
>.SELECT STATEMENT  
>...NESTED LOOPS  
>....TABLE ACCESS full dept  
>....TABLE ACCESS full emp
```

- Full Cartesian Product via Nested Loop Join (NLJ)
 - Init(RowSource1);
While not eof(RowSource1)
Loop Init(RowSource2);
While not eof(RowSource2)
Loop return(CurRec(RowSource1)+CurRec(RowSource2));
NxtRec(RowSource2);
End Loop;
NxtRec(RowSource1);
End Loop;

Two loops,
nested

Joins, Sort Merge (3.2)

```
SELECT *
FROM emp, dept
WHERE emp.d# = dept.d#;
```

```
>.SELECT STATEMENT
>...MERGE JOIN
>....SORT join
>.....TABLE ACCESS full emp
>....SORT join
>.....TABLE ACCESS full dept
```

- Inner Join, no indexes: Sort Merge Join (SMJ)

```
Tmp1 := Sort(RowSource1,JoinColumn);
Tmp2 := Sort(RowSource2,JoinColumn);
Init(Tmp1); Init(Tmp2);
While Sync(Tmp1,Tmp2,JoinColumn)
Loop return(CurRec(Tmp1)+CurRec(Tmp2));
End Loop;
```

Sync advances
pointer(s) to
next match

Joins (3.3)

```
SELECT *
FROM emp, dept
WHERE emp.d# = dept.d#;
```

Emp(d#)

```
>.SELECT STATEMENT
>...NESTED LOOPS
>....TABLE ACCESS full dept
>....TABLE ACCESS by rowid emp
>.....INDEX range scan e_emp_fk
```

- Inner Join, only **one side indexed**
 - NLJ starts with full scan of non-indexed table
 - Per row retrieved use index to find matching rows
 - Within 2nd loop a (current) value for d# is available!
 - And used to perform a range scan

Joins (3.4)

```
SELECT *
FROM emp, dept
WHERE emp.d# = dept.d#
```

Emp(d#)
Unique Dept(d#)

```
>.SELECT STATEMENT
>...NESTED LOOPS
>....TABLE ACCESS full dept
>....TABLE ACCESS by rowid emp
>.....INDEX range scan e_emp_fk
Or,
>.SELECT STATEMENT
>...NESTED LOOPS
>....TABLE ACCESS full emp
>....TABLE ACCESS by rowid dept
>.....INDEX unique scan e_dept_pk
```

- Inner Join, **both sides indexed**
 - RBO: NLJ, start with FTS of last table in FROM-clause
 - CBO: NLJ, start with FTS of biggest table in FROM-clause
 - Best multi-block I/O benefit in FTS
 - More likely smaller table will be in buffer cache

Joins (3.5)

```
SELECT *
FROM emp, dept
WHERE emp.d# = dept.d#
AND dept.loc = 'DALLAS'
```

Emp(d#)

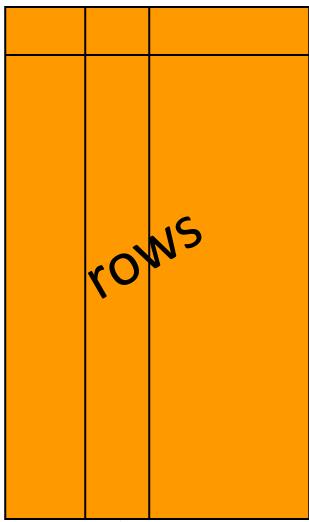
Unique Dept(d#)

```
>.SELECT STATEMENT
>...NESTED LOOPS
>....TABLE ACCESS full dept
>....TABLE ACCESS by rowid emp
>.....INDEX range scan e_emp_fk
```

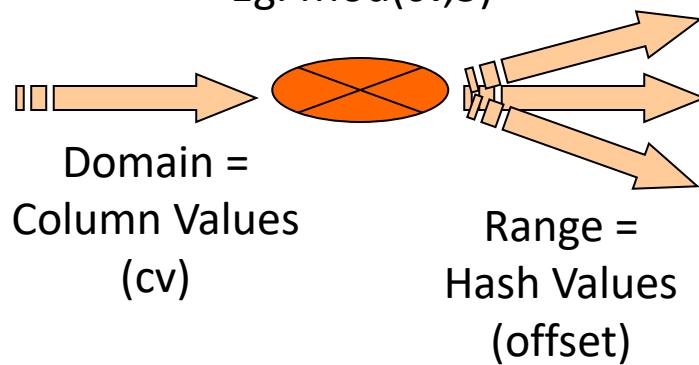
- Inner Join with additional conditions
 - Nested Loops
 - Always starts with table that has extra condition(s)

Hashing

Table

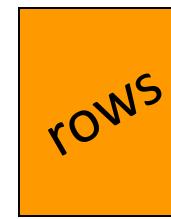
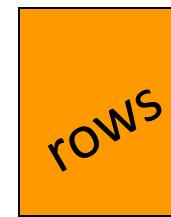
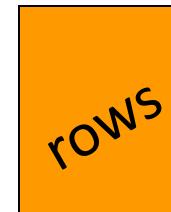


Hash Function
Eg. $\text{Mod}(cv,3)$



Equality search in where clause

```
SELECT *  
FROM table  
WHERE column = <value>
```



Buckets

Card. of range determines size of bucket

Joins, Hash (3.6)

```
SELECT *
FROM dept, emp
WHERE dept.d# = emp.d#
```

Emp(d#), Unique Dept(d#)

```
>.SELECT STATEMENT
>...HASH JOIN
>....TABLE ACCESS full dept
>....TABLE ACCESS full emp
```

- Tmp1 := Hash(RowSource1,JoinColumn); -- In memory
Init(RowSource2);
While not eof(RowSource2)
Loop HashInit(Tmp1,JoinValue); -- Locate bucket
 While not eof(Tmp1)
 Loop return(CurRec(RowSource2)+CurRec(Tmp1));
 NxtHashRec(Tmp1,JoinValue);
 End Loop; NxtRec(RowSource2);
 End Loop;

Joins, Hash (3.6)

- Must be explicitly enabled via init.ora file:
 - Hash_Join_Enabled = True
 - Hash_Area_Size = <bytes>
- If hashed table **does not fit in memory**
 - 1st rowsource: temporary hash cluster is built
 - And written to disk (I/O's) in partitions
 - 2nd rowsource also converted using same hash-function
 - Per 'bucket' rows are matched and returned
 - One bucket must fit in memory, else very bad performance

Subquery (4.1)

```
SELECT dname, deptno
  FROM dept
 WHERE d# IN
 (SELECT d#
   FROM emp);
```

```
>.SELECT STATEMENT
>...NESTED LOOPS
>....VIEW
>.....SORT unique
>.....TABLE ACCESS full emp
>.....TABLE ACCESS by rowid dept
>.....INDEX unique scan i_dept_pk
```

- Transformation into join
 - **Temporary view is built** which drives the nested loop

Subquery, Correlated (4.2)

```
SELECT *
  FROM emp e
 WHERE sal >
    (SELECT sal
      FROM emp m
     WHERE m.e#=e.mgr#)
```

```
>.SELECT STATEMENT
>...FILTER
>....TABLE ACCESS full emp
>....TABLE ACCESS by rowid emp
>.....INDEX unique scan i_emp_pk
```

- “Nested Loops”-like FILTER
 - For each row of 1st rowsource, execute 2nd rowsource and filter on truth of subquery-condition
 - Subquery can be re-written as self-join of EMP table

Subquery, Correlated (4.2)

```
SELECT *
  FROM emp e, emp m
 WHERE m.e#=e.mgr#
   AND e.sal > m.sal;
```

```
>.SELECT STATEMENT
>...NESTED LOOPS
>....TABLE ACCESS full emp
>....TABLE ACCESS by rowid emp
>.....INDEX unique scan i_emp_pk
```

- Subquery **rewrite to join**
 - Subquery can also be rewritten to EXISTS-subquery

Subquery, Correlated (4.2)

```
SELECT *
  FROM emp e
 WHERE exists
    (SELECT 'less salary'
      FROM emp m
     WHERE e.mgr# = m.e#
       and m.sal < e.sal);
```

```
>.SELECT STATEMENT
>...FILTER
>....TABLE ACCESS full emp
>....TABLE ACCESS by rowid emp
>.....INDEX unique scan i_emp_pk
```

- Subquery rewrite to EXISTS query
 - For each row of 1st rowsource, execute 2nd rowsource
And filter on retrieval of rows by 2nd rowsource

Concatenation (4.3)

```
SELECT *
  FROM emp
 WHERE mgr# = 100
   OR job = 'CLERK';
```

Emp(mgr#)
Emp(job)

```
>.SELECT STATEMENT
>...CONCATENATION
>....TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_m
>....TABLE ACCESS by rowid emp
>.....INDEX range scan i_emp_j
```

- **Concatenation (OR-processing)**
 - Similar to query **rewrite** into 2 separate queries
 - Which are then ‘concatenated’
 - If one index was missing => Full Table Scan

Inlist Iterator (4.4)

```
SELECT *
FROM dept
WHERE d# in (10,20,30);
```

Unique Dept(d#)

```
>.SELECT STATEMENT
>...INLIST ITERATOR
>....TABLE ACCESS by rowid dept
>.....INDEX unique scan i_dept_pk
```

- Iteration over enumerated value-list
 - Every value executed separately
- Same as concatenation of 3 “OR-red” values

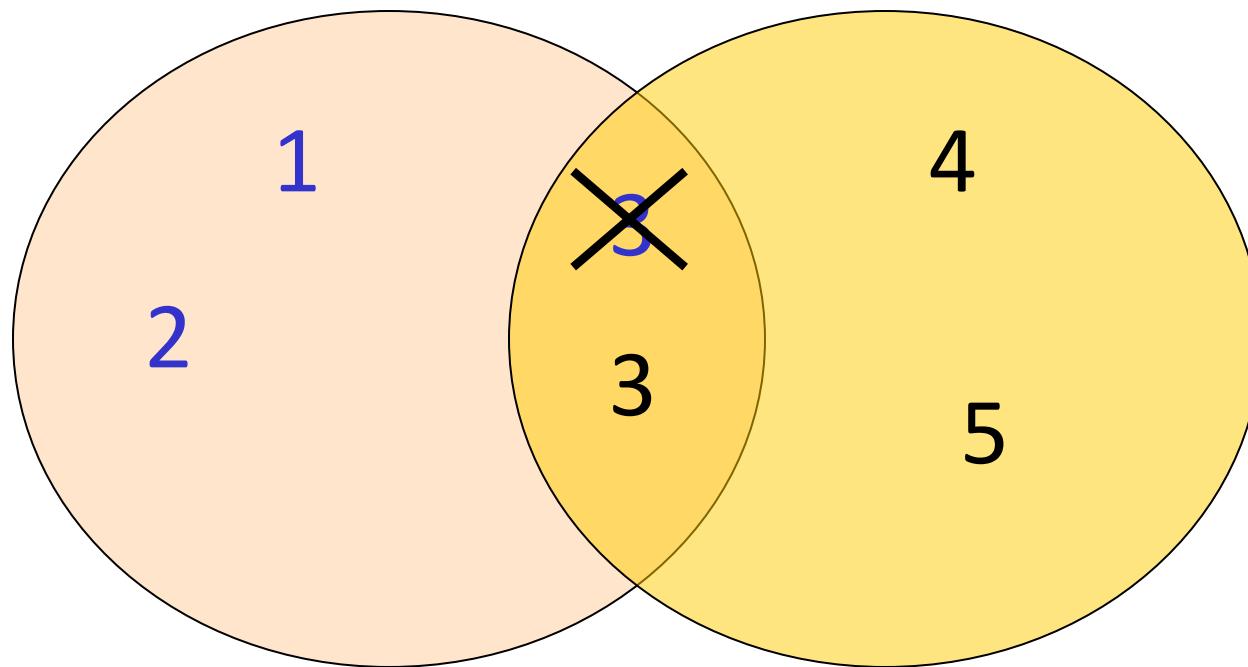
Union (4.5)

```
SELECT empno  
FROM emp  
UNION  
SELECT deptno  
FROM dept;
```

```
>.SELECT STATEMENT  
>...SORT unique  
>....UNION  
>.....TABLE ACCESS full emp  
>.....TABLE ACCESS full dept
```

- Union followed by **Sort-Unique**
 - Sub rowsources are all executed/optimized individually
 - Rows retrieved are ‘concatenated’
 - **Set theory** demands **unique elements** (Sort)

UNION



Union All (4.6)

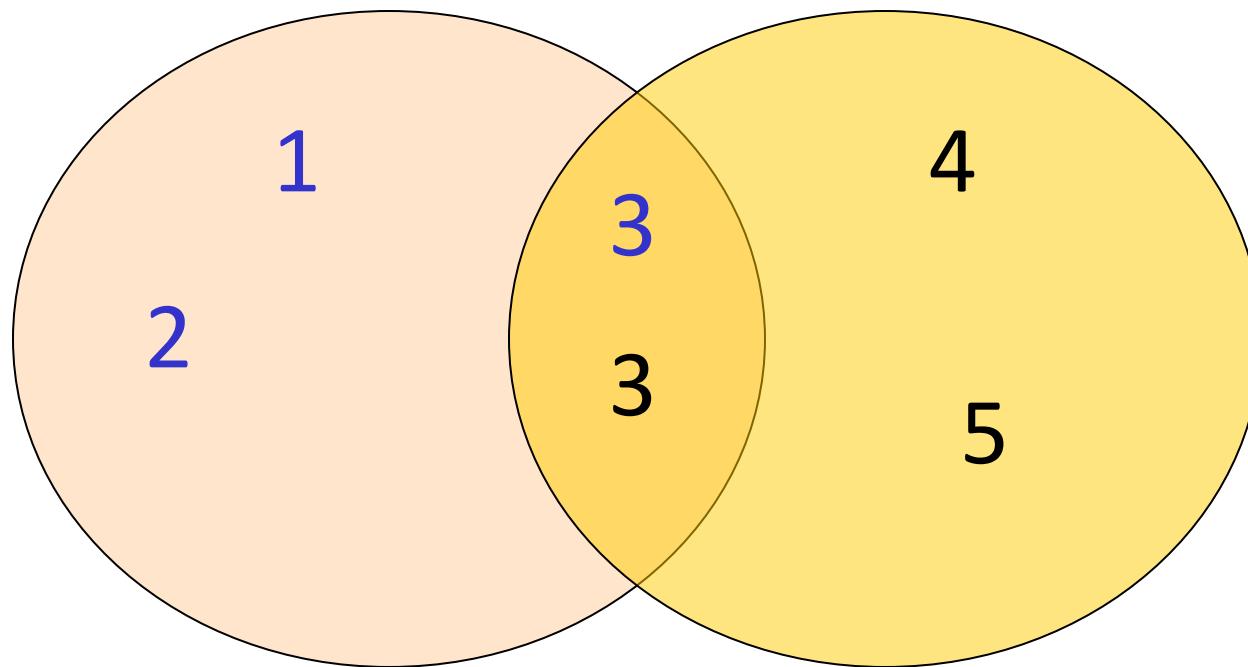
```
SELECT empno  
FROM emp  
UNION ALL  
SELECT deptno  
FROM dept;
```

```
>.SELECT STATEMENT  
>...UNION-ALL  
>....TABLE ACCESS full emp  
>....TABLE ACCESS full dept
```

- **Union-All:** result is a '**bag**', not a set
 - (expensive) Sort-operator not necessary

Use UNION-ALL if you know the bag is a set.
(saving an expensive sort)

UNION ALL



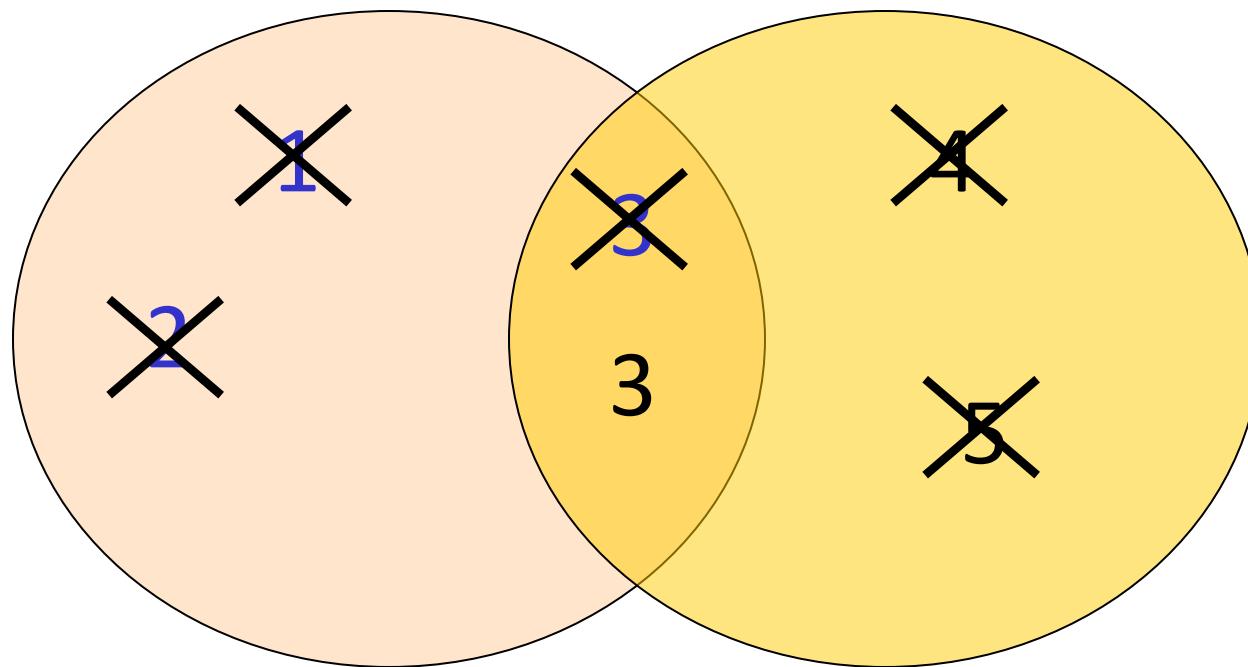
Intersect (4.7)

```
SELECT empno  
FROM emp  
INTERSECT  
SELECT deptno  
FROM dept;
```

```
>.SELECT STATEMENT  
>...INTERSECTION  
>....SORT unique  
>.....TABLE ACCESS full emp  
>....SORT unique  
>.....TABLE ACCESS full dept
```

- INTERSECT
 - Sub rowsources are all executed/optimized individually
 - Very **similar to Sort-Merge-Join** processing
 - Full rows are sorted and matched

INTERSECT



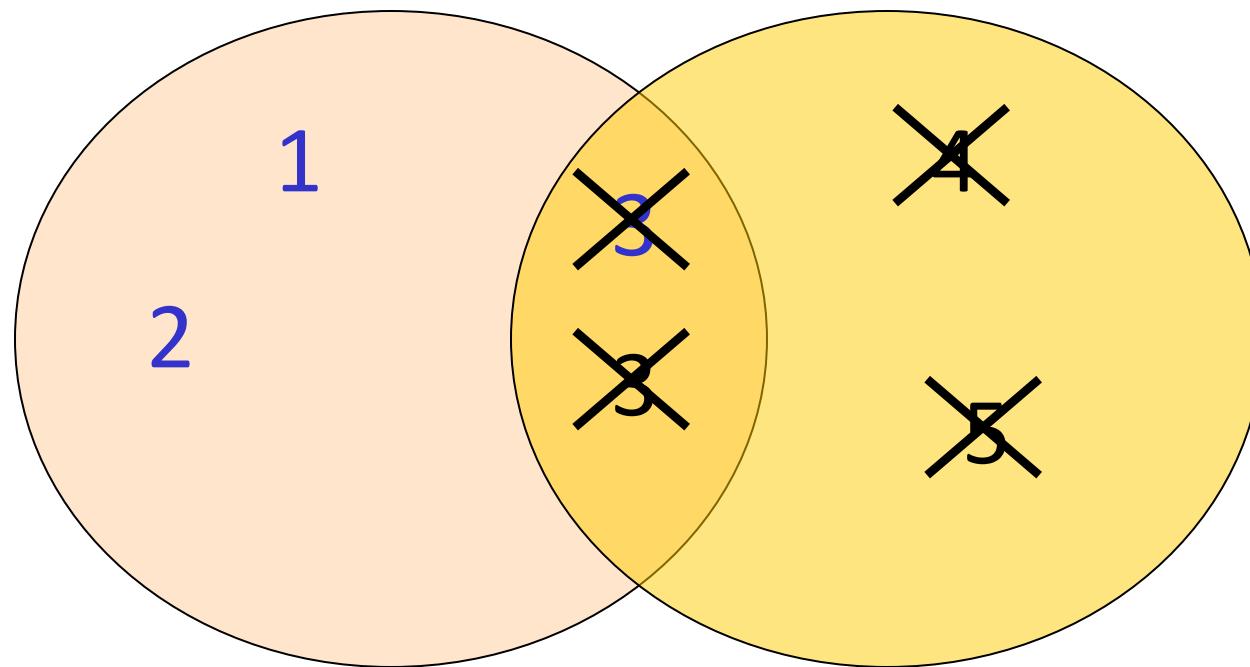
Minus (4.8)

```
SELECT empno  
FROM emp  
MINUS  
SELECT deptno  
FROM dept;
```

```
>.SELECT STATEMENT  
>...MINUS  
>....SORT unique  
>.....TABLE ACCESS full emp  
>....SORT unique  
>.....TABLE ACCESS full dept
```

- MINUS
 - Sub rowsources are all executed/optimized individually
 - Similar to **INTERSECT** processing
 - Instead of match-and-return, match-and-exclude

MINUS



Utilities

- Tracing
- SQL Hints
- Analyze command
- Dbms_Stats package

Trace Files

- Explain-plan: give insight before execution
- Tracing: give insight in actual execution
 - CPU-time spent
 - Elapsed-time
 - # of physical block-I/O's
 - # of cached block-I/O's
 - Rows-processed per row-source
- Session must be put in trace-mode
 - Alter session set sql_trace=true;
 - Exec dbms_system.set_sql_trace_in_session(sid,s#,T/F);

Trace Files

- Tracefile is generated **on database server**
 - Needs to be formatted with TKPROF-utility

```
tkprof <tracefile> <tkp-file> <un>/<pw>
```

- Two sections per SQL-statement:

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.06	0.07	0	0	0	0
Execute	1	0.01	0.01	0	0	0	0
Fetch	1	0.11	0.13	0	37	2	2
total	3	0.18	0.21	0	37	2	2

Trace Files

- 2nd section: **extended explain plan:**

- Example 4.2 (emp with more sal than mgr),

```
#R Plan
2 SELECT STATEMENT
14 FILTER
14 TABLE ACCESS (FULL) OF 'EMP'
11 TABLE ACCESS (BY ROWID) OF 'EMP'
12 INDEX (UNIQUE SCAN) OF 'I_EMP_PK' (UNIQUE)
```

- Emp has 14 records
- **Two of them** have no manager (NULL mgr column value)
- **One of them** points to non-existing employee
- **Two actually** earn more than their manager

Hints

- Force optimizer to pick specific alternative
 - Implemented via embedded comment

```
SELECT /*+ <hint> */ ....  
FROM ....  
WHERE ....
```

```
UPDATE /*+ <hint> */ ....  
WHERE ....
```

```
DELETE /*+ <hint> */ ....  
WHERE ....
```

INSERT (see SELECT)

Hints

- Common hints

- Full(<tab>)
- Index(<tab> <ind>)
- Index_asc(<tab> <ind>)
- Index_desc(<tab> <ind>)
- Ordered
- Use_NL(<tab> <tab>)
- Use_Merge(<tab> <tab>)
- Use_Hash(<tab> <tab>)
- Leading(<tab>)
- First_rows, All_rows, Rule

Analyze command

- Statistics need to be periodically generated
 - Done via ‘ANALYZE’ command

```
Analyze <Table | Index> <x>  
<compute | estimate | delete> statistics  
<sample <x> <Rows | Percent>>
```

```
Analyze table emp estimate statistics sample 30 percent;
```

ANALYZE will be de-supported

Dbms_Stats Package

- Successor of Analyze command

- Dbms_stats.gather_index_stats(<owner>,<index>,<blocksample>,<est.percent>)
- Dbms_stats.gather_table_stats(<owner>,<table>,<blocksample>,<est.percent>)
- Dbms_stats.delete_index_stats(<owner>,<index>)
- Dbms_stats.delete_table_stats(<owner>,<table>)

```
SQL>exec dbms_stats.gather_table_status('scott','emp',null,30);
```

Warehouse Specifics

- Traditional Star Query
- Bitmap Indexes
 - Bitmap merge, and, conversion-to-rowid
 - Single table query
- Star Queries
 - Multiple tables

Traditional Star Query

```
SELECT f.*  
FROM a,b,f  
WHERE a.pk = f.a_fk  
AND b.pk = f.b_fk  
AND a.t = ... AND b.s = ...
```

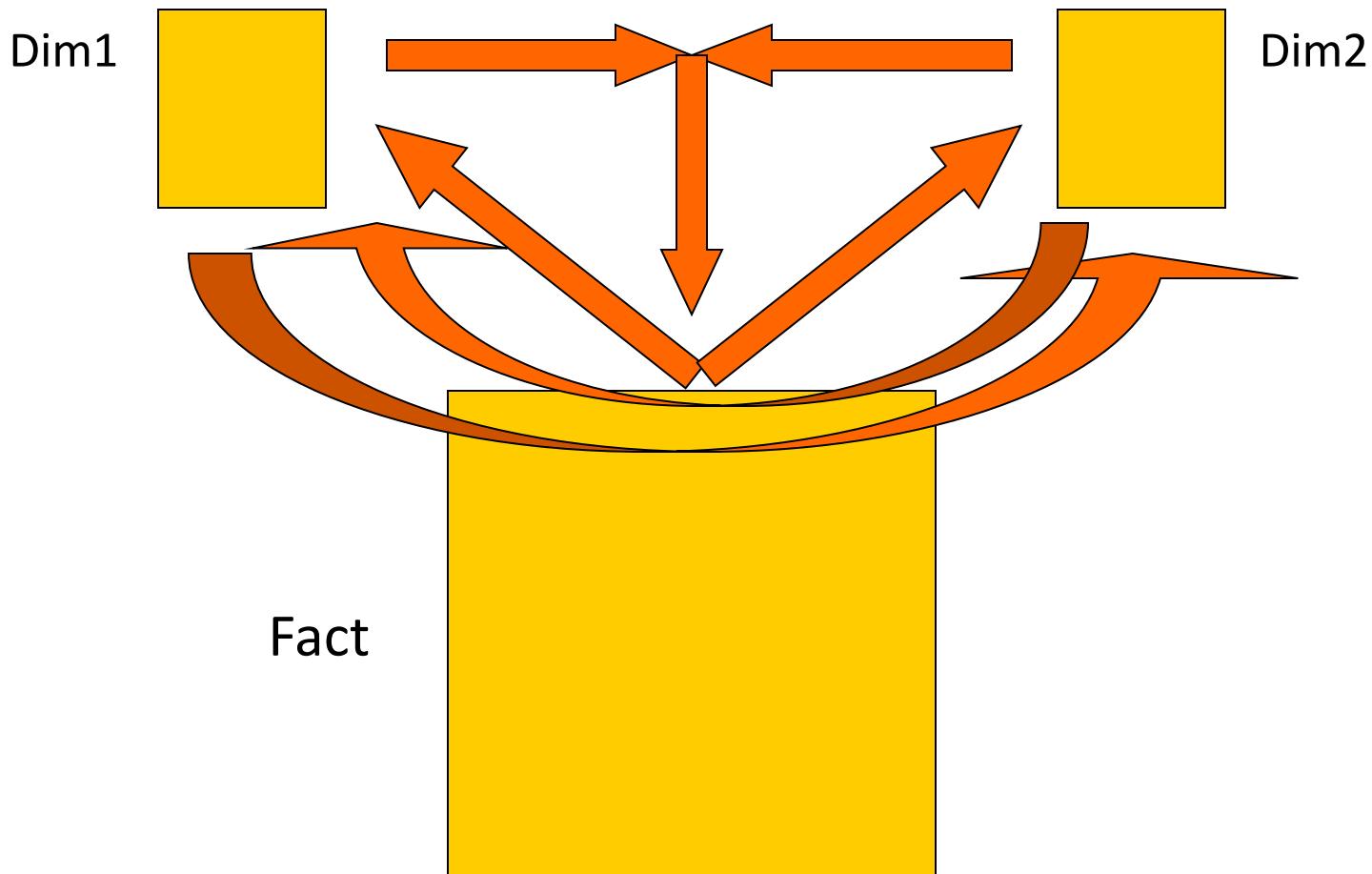
A(pk), B(pk)
F(a_fk), F(b_fk)

```
>.SELECT STATEMENT  
>...NESTED LOOPS  
>....NESTED LOOPS  
>.....TABLE ACCESS full b  
>.....TABLE ACCESS by rowid fact  
>.....INDEX range scan i_fact_b  
>.....TABLE ACCESS by rowid a  
>.....INDEX unique scan a_pk
```

- Double nested loops
 - Pick one table as start (A or B)
 - Then follow join-conditions using Nested_Loops

Too complex for AND-EQUAL

Traditional Star Query



Four access-order alternatives!

Traditional Star Query

```
SELECT f.*  
FROM a,b,f  
WHERE a.pk = f.a_fk  
AND b.pk = f.b_fk  
AND a.t = ... AND b.s = ...  
F(a_fk,b_fk,...)
```

```
>.SELECT STATEMENT  
>...NESTED LOOPS  
>....MERGE JOIN cartesian  
>.....TABLE ACCESS full a  
>.....SORT join  
>.....TABLE ACCESS full b  
>.....TABLE ACCESS by rowid fact  
>.....INDEX range scan I_f_abc
```

- Concatenated Index Range Scans for Star Query
 - At least two dimensions
 - Index at least one column more than dimensions used
 - Merge-Join-Cartesian gives all applicable dimension combinations
 - Per combination the concatenated index is probed

Bitmap Access, Single Table

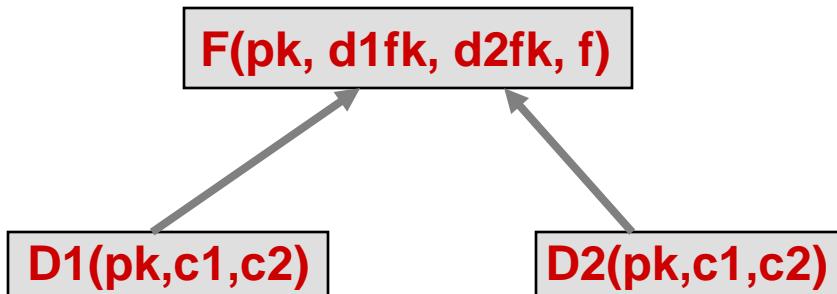
```
SELECT count(*)  
FROM customer  
WHERE status='M'  
AND region in ('C','W');
```

```
>.....TABLE ACCESS (BY INDEX ROWID) cust  
>.....BITMAP CONVERSION to rowids  
>.....BITMAP AND  
>.....BITMAP INDEX single value cs  
>.....BITMAP MERGE  
>.....BITMAP KEY ITERATION  
>.....BITMAP INDEX range scan cr
```

- Bitmap OR's, AND's and CONVERSION
 - Find Central and West bitstreams (bitmap key-iteration)
 - Perform logical OR on them (bitmap merge)
 - Find Married bitstream
 - Perform logical AND on region bitstream (bitmap and)
 - Convert to actual rowid's
 - Access table

Bitmap Access, Star Query

Bitmap indexes: id1, id2



```
SELECT sum(f)
FROM F,D1,D2
WHERE F=D1 and F=D2
AND D1.C1=<...>
AND D2.C2=<...>
```

```
>..... TABLE ACCESS (BY INDEX ROWID) f
>..... BITMAP CONVERSION (TO ROWIDS)
>..... BITMAP AND
>..... BITMAP MERGE
>..... BITMAP KEY ITERATION
>..... TABLE ACCESS (FULL) d1
>..... BITMAP INDEX (RANGE SCAN) id1
>..... BITMAP MERGE
>..... BITMAP KEY ITERATION
>..... TABLE ACCESS (FULL) d2
>..... BITMAP INDEX (RANGE SCAN) id2
```

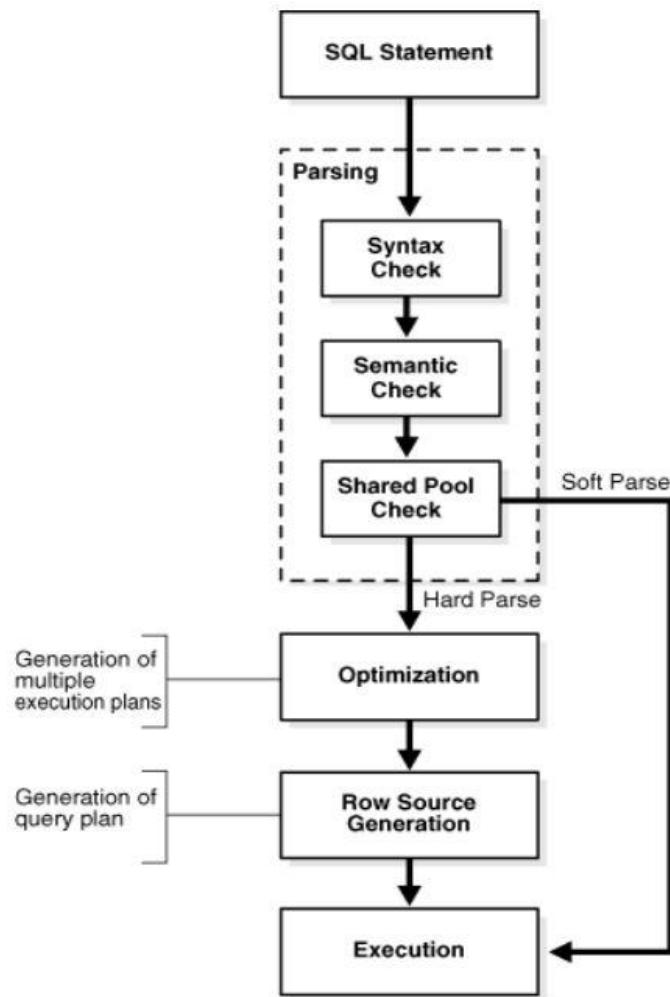
Warehouse Hints

- Specific star-query related hints
 - Star
 - Traditional: via concat-index range scan
 - Star_transformation
 - Via single column bitmap index merges/and's
 - Fact(t) / No_fact(t)
 - Help star_transformation
 - Index_combine(t i1 i2 ...)
 - Explicitely instruct which indexes to merge/and

SQL Tuning: Roadmap

- Able to read plan
- Able to translate plan into 3GL program
 - Know your row-source operators
- Able to read SQL
- Able to translate SQL into business query
 - Know your datamodel
- Able to judge outcome
 - Know your business rules / data-statistics
 - Better than CBO does
- Experts:
 - Optimize SQL while writing SQL...

Stages of SQL processing:



Syntax Check

Oracle Database must check each SQL statement for **syntactic validity**.

Semantic Check

The semantics of a statement are its meaning. A semantic check determines whether a statement is meaningful, for example, **whether the objects and columns in the statement exist**.

Shared Pool Check

During the parse, the database performs a shared pool check to determine whether it can skip resource-intensive steps of statement processing. To this end, the database uses a hashing algorithm to generate a hash value for every SQL statement. When a user submits a SQL statement, the database searches the **shared SQL area** to see if **an existing parsed statement** has the same hash value.

Hard parse

If Oracle Database cannot reuse existing code, then it must **build a new executable version** of the application code. This operation is known as a **hard parse**, or a **library cache miss**.

Soft parse

A **soft parse** is any parse that is not a hard parse. If the submitted statement is the same as a reusable SQL statement in the shared pool, then Oracle Database **reuses the existing code**. This reuse of code is also called a **library cache hit**. If a check determines that a statement in the shared pool has the same hash value, then the database performs semantic and environment checks to determine whether the statements have the same meaning. **Identical syntax is not enough**. Even if two statements are semantically identical, **an environmental**

difference can force a hard parse. In this context, the **optimizer environment** is the totality of session settings that can affect execution plan generation, such as the work area size or optimizer settings (for example, the optimizer mode).

SQL Optimization

During the optimization stage, Oracle Database must perform a hard parse at least once for every unique DML statement and performs the optimization during this parse.

SQL Row Source Generation

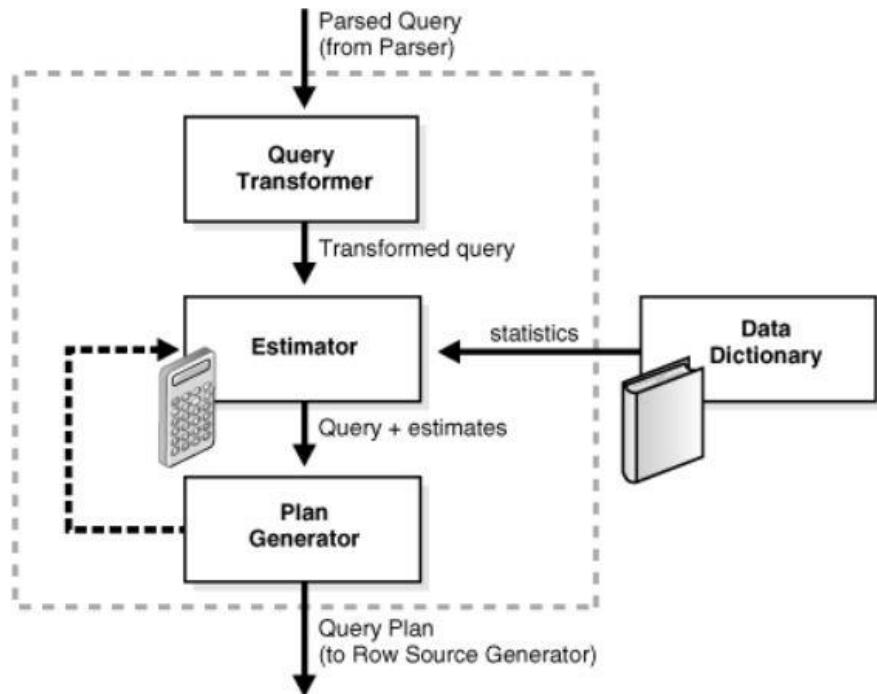
The **row source generator** is software that receives the optimal execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database.

The iterative plan is a **binary program** that, when executed by the SQL engine, produces the result set. The plan takes the form of a combination of steps. **Each step returns a row set**. The next step either uses the rows in this set, or the last step returns the rows to the application issuing the SQL statement.

SQL Execution

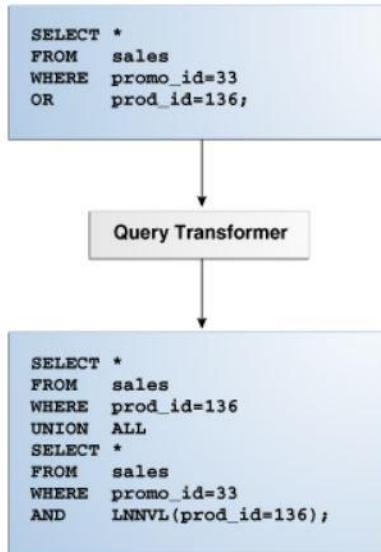
During execution, the **SQL engine executes each row source in the tree** produced by the row source generator.

Optimizer components:



Query Transformer

For some statements, the query transformer determines whether it is advantageous to rewrite the original SQL statement into a semantically equivalent SQL statement with a lower cost.



LNNVL function (“logical not null value”)

Tink of it as an **opposite detector**: see details in documentation)

It takes as an argument a condition and returns TRUE if the condition is FALSE or UNKNOWN and FALSE if the condition is TRUE.

Query Transformations

The optimizer employs several query transformation techniques.

In **OR expansion**, the optimizer transforms a query with a WHERE clause containing **OR** operators into a query that uses the **UNION ALL** operator.

In **view merging**, the optimizer merges the query block representing a view into the query block that contains it. View merging can improve plans by enabling the optimizer to consider additional join orders, access methods, and other transformations.

In **simple view merging**, the optimizer merges select-project-join views.

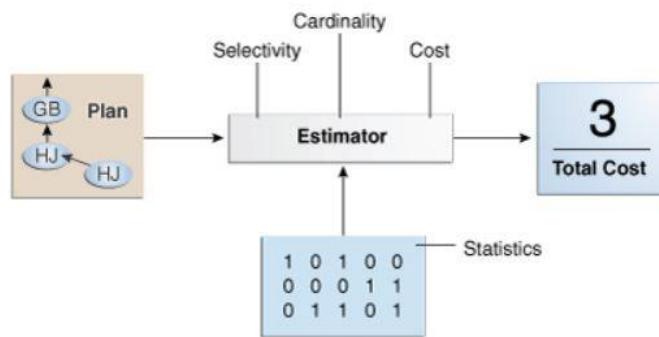
In **complex view merging**, the optimizer merges views containing GROUP BY and DISTINCT views. Like simple view merging, complex merging enables the optimizer to consider additional join orders and access paths.

In **subquery unnesting**, the optimizer transforms a nested query into an equivalent join statement, and then optimizes the join.

In the cost-based transformation known as **join factorization**, the optimizer can factorize common computations from branches of a UNION ALL query.

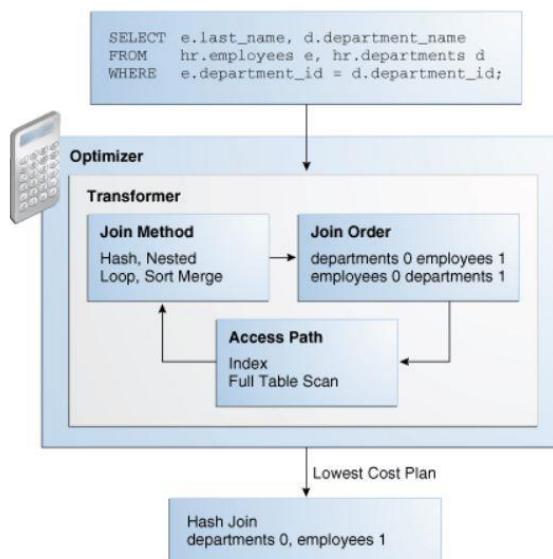
Estimator

The **estimator** is the component of the optimizer that determines the overall cost of a given execution plan.



Plan generator

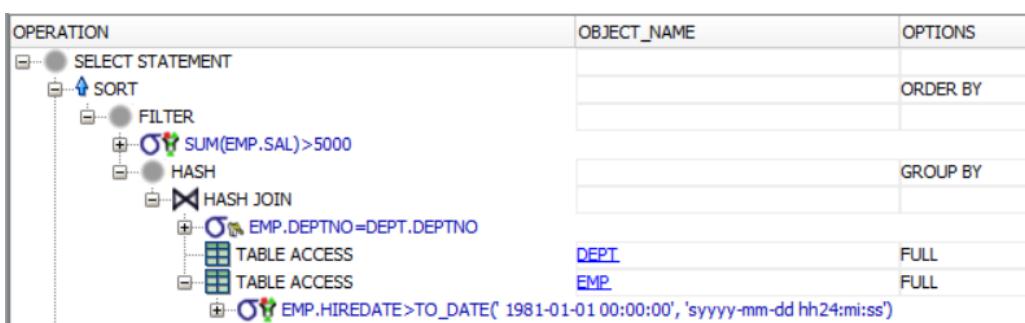
The **plan generator** explores various plans for a query block by trying out **different access paths, join methods, and join orders**. Many plans are possible because of the various combinations that the database can use to produce the same result. The **optimizer picks** the plan with the **lowest cost**.



In Oracle Database, **adaptive query optimization** enables the optimizer to make run-time adjustments to execution plans and discover additional information that can lead to better statistics.

Example:

```
SELECT dname, job, AVG(sal)
FROM nikovits.emp NATURAL JOIN nikovits.dept
WHERE hiredate > to_date('1981.01.01', 'yyyy.mm.dd')
GROUP BY dname, job HAVING SUM(sal) > 5000
ORDER BY AVG(sal) DESC;
```



Steps and result sets of the previous execution plan:

TMP1 := $\Pi_{JOB, SAL, DEPTNO} (\sigma_{Hiredate > \dots} EMP)$ **EMP FULL**

JOB	SAL	DEPTNO
SALESMAN	1600	30
SALESMAN	1250	30
MANAGER	2975	20
MANAGER	2450	10
...

TMP2 := $\Pi_{DEPTNO, DNAME} DEPT$ **DEPT FULL**

DEPTNO	DNAME
10	ACCOUNTING
20	RESEARCH
...	...

TMP3 := $\Pi_{DNAME, JOB, SAL} TMP1 \bowtie TMP2$ **HASH JOIN**

DNAME	JOB	SAL
SALES	SALESMAN	1600
SALES	SALESMAN	1250
RESEARCH	MANAGER	2975
ACCOUNTING	MANAGER	2450
RESEARCH	ANALYST	3000
...

TMP4 := $\gamma_{DNAME, JOB, COUNT(SAL), SUM(SAL)} TMP3$ **GROUP BY**

DNAME	JOB	COUNT(SAL)	SUM(SAL)
SALES	SALESMAN	4	5600
RESEARCH	ANALYST	2	6000
MARKETING	SALESMAN	1	1600
ACCOUNTING	MANAGER	1	2450
ACCOUNTING	PRESIDENT	1	5000
...

TMP5 := $\Pi_{DNAME, JOB, SumSal/Cnt \rightarrow AvgSal} (\sigma_{SumSal > 5000} TMP4)$ **FILTER**

DNAME	JOB	AVG(SAL)
SALES	SALESMAN	1400
RESEARCH	ANALYST	3000

RESULT := $\tau_{AvgSal} TMP5$ **SORT**

DNAME	JOB	AVG(SAL)
RESEARCH	ANALYST	3000
SALES	SALESMAN	1400

Ullman et al. : Database System Principles

Notes 08: Failure Recovery

Integrity or correctness of data

- Would like data to be “accurate” or “correct” at all times

EMP

Name	Age
White	52
Green	3421
Gray	1

Integrity or consistency **constraints**

- **Predicates** data must satisfy
- Examples:
 - x is key of relation R
 - $x \rightarrow y$ (func. dependency) holds in R
 - $\text{Domain}(x) = \{\text{Red, Blue, Green}\}$
 - no employee should make more than twice the average salary

Definition:

- Consistent state: satisfies all constraints
- Consistent DB: DB in consistent state

Constraints (as we use here) **may**
not capture “full correctness”

Example 1 Transaction constraints

- When salary is updated,
 new salary > old salary
- When account record is deleted,
 balance = 0

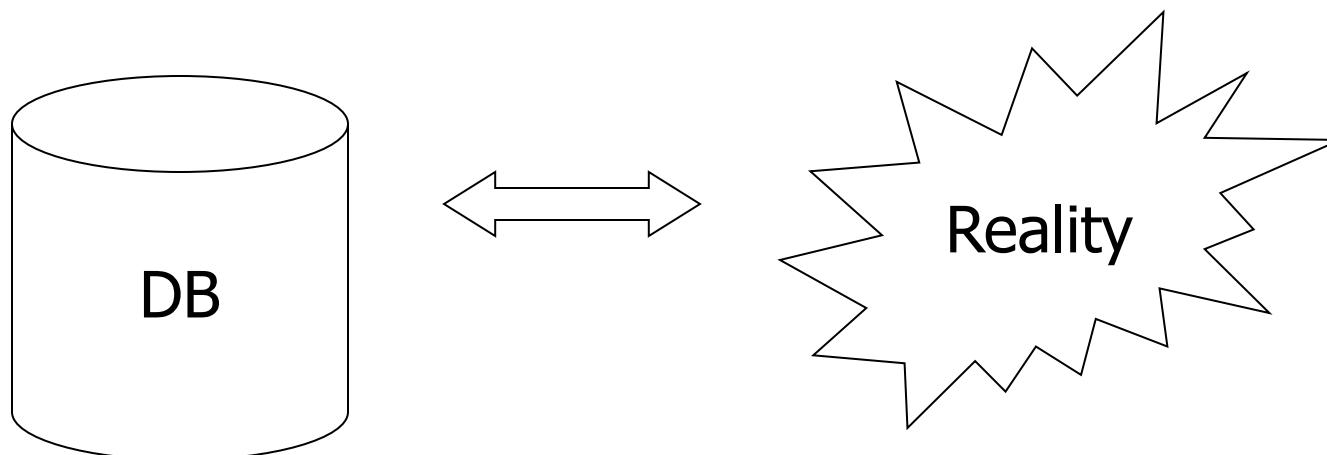
Note: could be “emulated” by simple constraints, e.g.,

if deleted = true, then balance = 0

account	Acct #	balance	deleted?
---------	--------	------	---------	----------

Constraints (as we use here) may
not capture “full correctness”

Example 2 Database should reflect
real world



→ in any case, continue with constraints...

Observation: DB cannot be consistent
always!

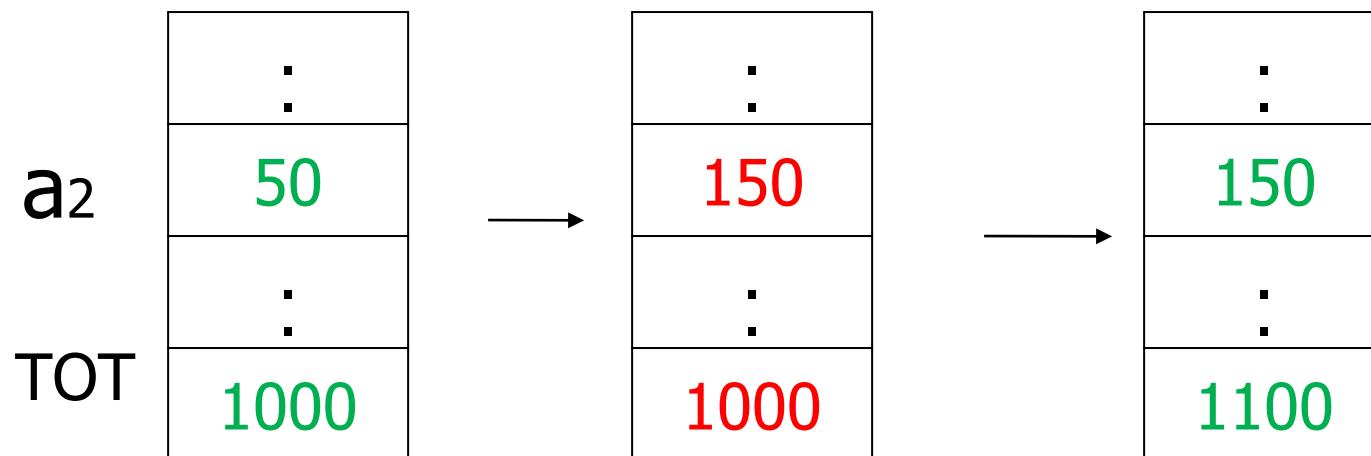
Example: $a_1 + a_2 + \dots + a_n = \text{TOT}$ (constraint)

Deposit \$100 in a_2 : $\begin{cases} a_2 \leftarrow a_2 + 100 \\ \text{TOT} \leftarrow \text{TOT} + 100 \end{cases}$

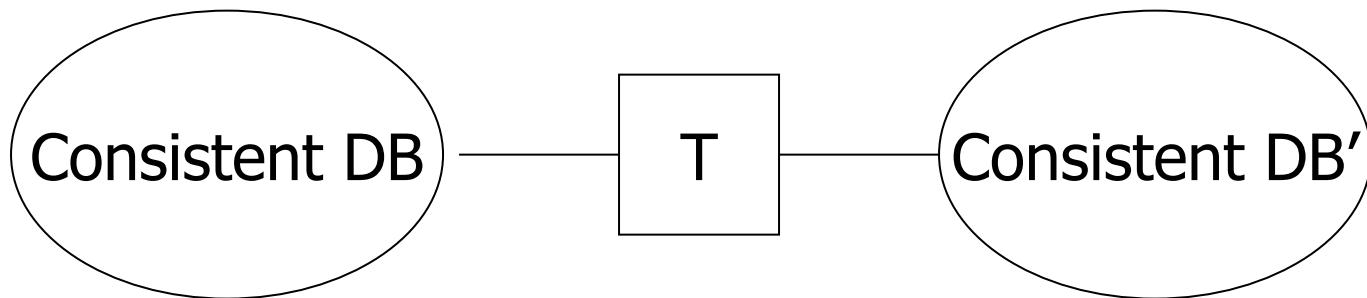
Example: $a_1 + a_2 + \dots + a_n = \text{TOT}$ (**constraint**)

Deposit \$100 in a_2 : $a_2 \leftarrow a_2 + 100$

$\text{TOT} \leftarrow \text{TOT} + 100$



Transaction: collection of actions
that preserve consistency



Big assumption:

If T starts with consistent state +

T executes in isolation

⇒ T leaves consistent state

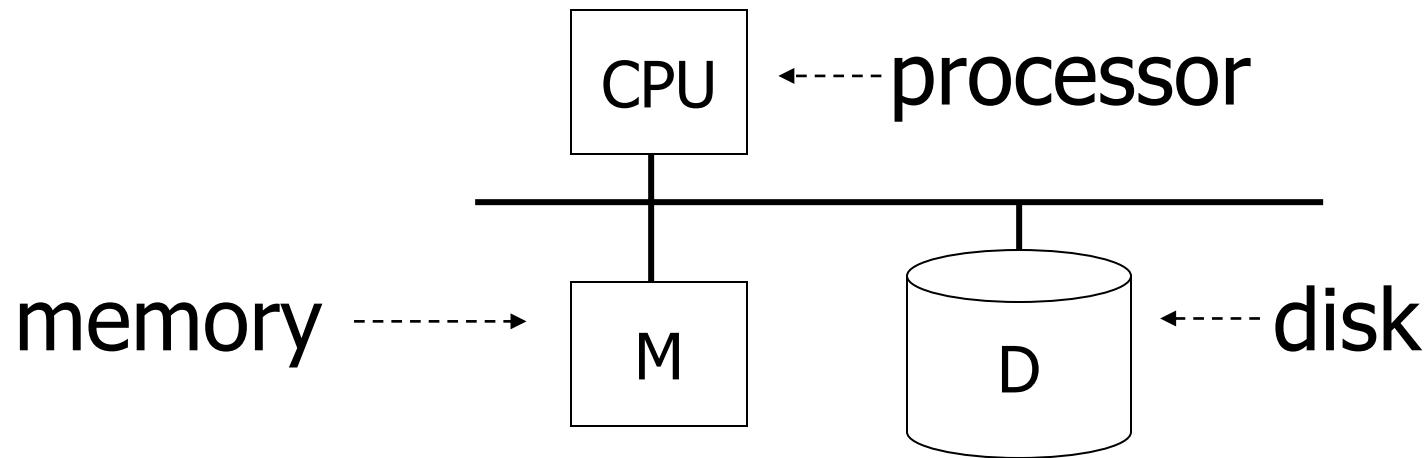
Why?

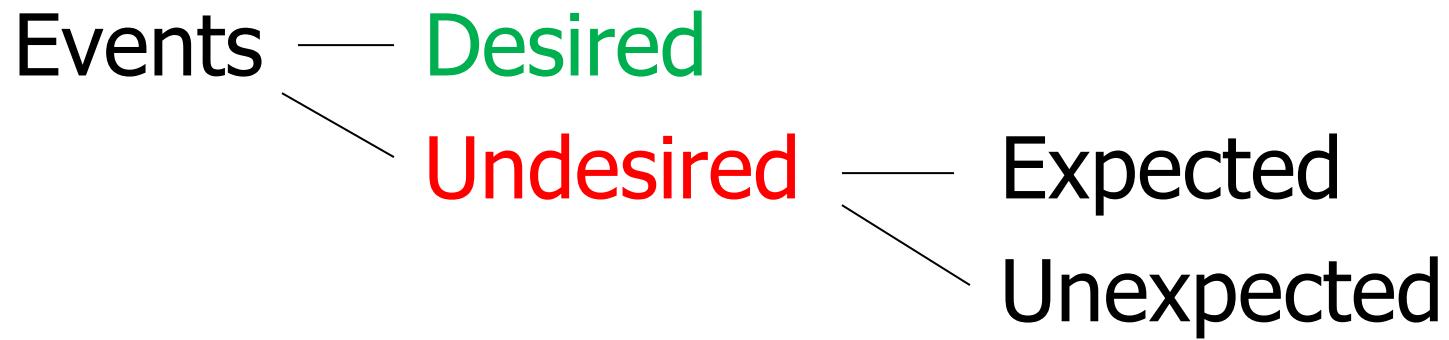
Constraints are enforced by the database, so any transaction that violates them will be rejected by the DBMS.

How can constraints be violated?

- Transaction bug (incomplete transaction)
- DBMS bug (some process)
- Hardware failure
 - e.g., disk crash alters balance of account
- Data sharing
 - e.g.: T1: give 10% raise to programmers
 - T2: change programmers \Rightarrow systems analysts

Our failure model





Desired events: see product manuals....

Undesired expected events:

System crash

- memory lost
- cpu halts, resets

that's it!!

Undesired Unexpected: **Everything else!**

Undesired Unexpected: **Everything else!**

Examples:

- Disk data is lost
- Memory lost without CPU halt
- CPU implodes wiping out universe....

Is this model reasonable?

Approach: Add low level checks + redundancy to increase probability model holds

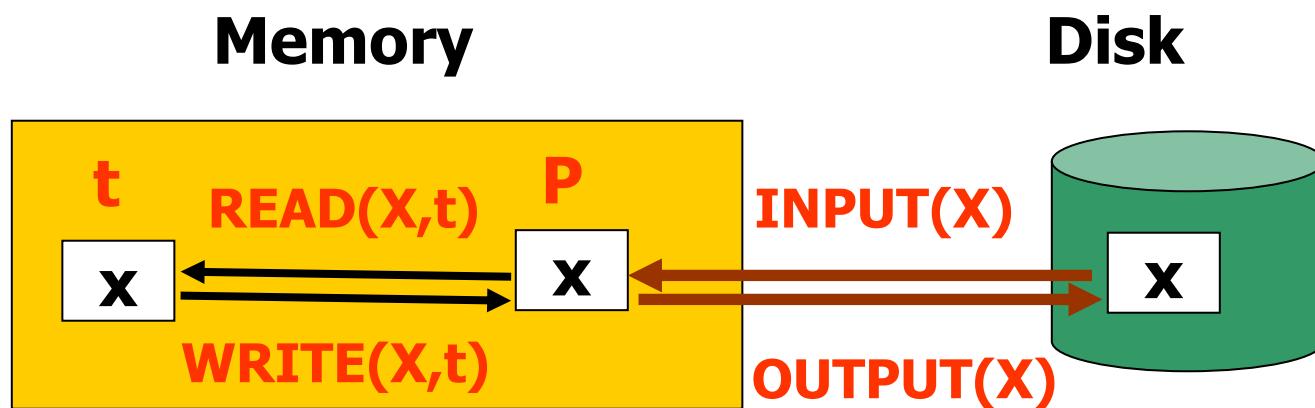
E.g., { Replicate disk storage (**RAID**)
 { Memory **parity**
 { CPU checks

The primitive operations of Transactions

There are 3 important address spaces:

1. The disk blocks
2. The shared main memory
3. The local address space of a Transaction

Basic operations



Operations:

- Input (x): block containing x → memory
- Output (x): block containing x → disk
- Read (x,t): do input(x) if necessary
 $t \leftarrow$ value of x in block
- Write (x,t): do input(x) if necessary
value of x in block $\leftarrow t$

Our simple example transaction

Example

Constraint: $A=B$

$T_1: A \leftarrow A \times 2$

$B \leftarrow B \times 2$

Steps of a transaction and its effect on memory and disk

• <i>Action</i>	<i>t</i>	<i>M-A</i>	<i>M-B</i>	<i>D-A</i>	<i>D-B</i>
1. READ (A, t)	8	8		8	8
2. $t := t*2$	16	8		8	8
3. WRITE (A, t)	16	16		8	8
4. READ (B, t)	8	16	8	8	8
5. $t := t*2$	16	16	8	8	8
6. WRITE (B, t)	16	16	16	8	8
7. OUTPUT (A)	16	16	16	16	8
8. OUTPUT (B)	16	16	16	16	16

Key problem

Unfinished transaction

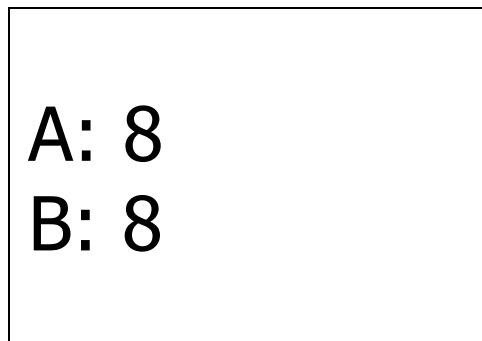
Example

Constraint: $A=B$

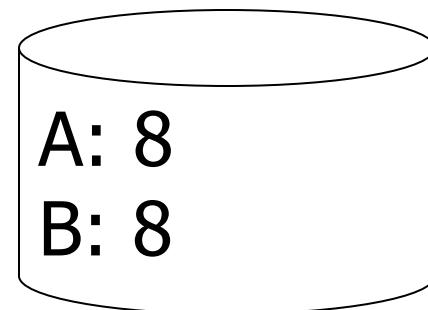
$T_1: A \leftarrow A \times 2$

$B \leftarrow B \times 2$

```
T1: Read (A,t); t ← t×2
    Write (A,t);
    Read (B,t); t ← t×2
    Write (B,t);
    Output (A);
    Output (B);
```

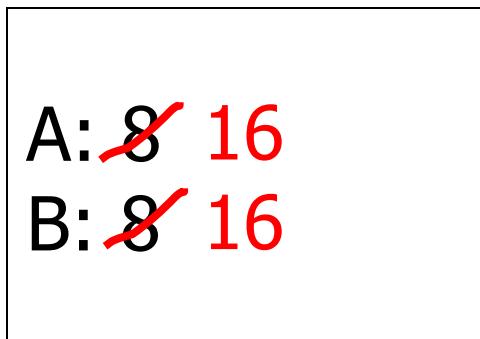


memory

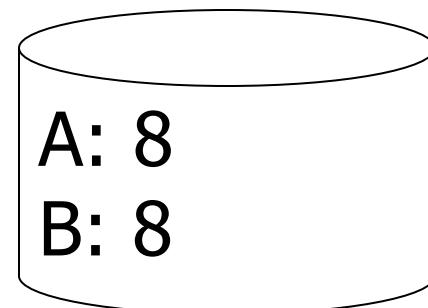


disk

```
T1: Read (A,t); t ← t×2
    Write (A,t);
    Read (B,t); t ← t×2
    Write (B,t);
    Output (A);
    Output (B);
```



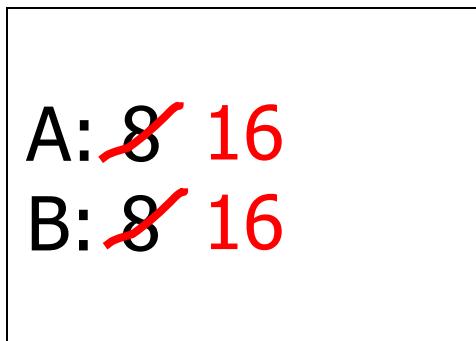
memory



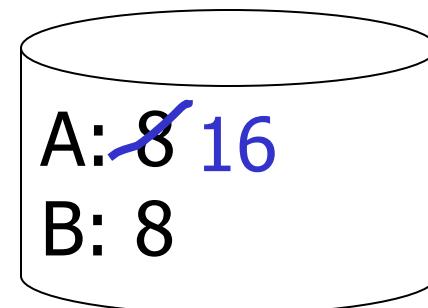
disk

T1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);

failure!



memory



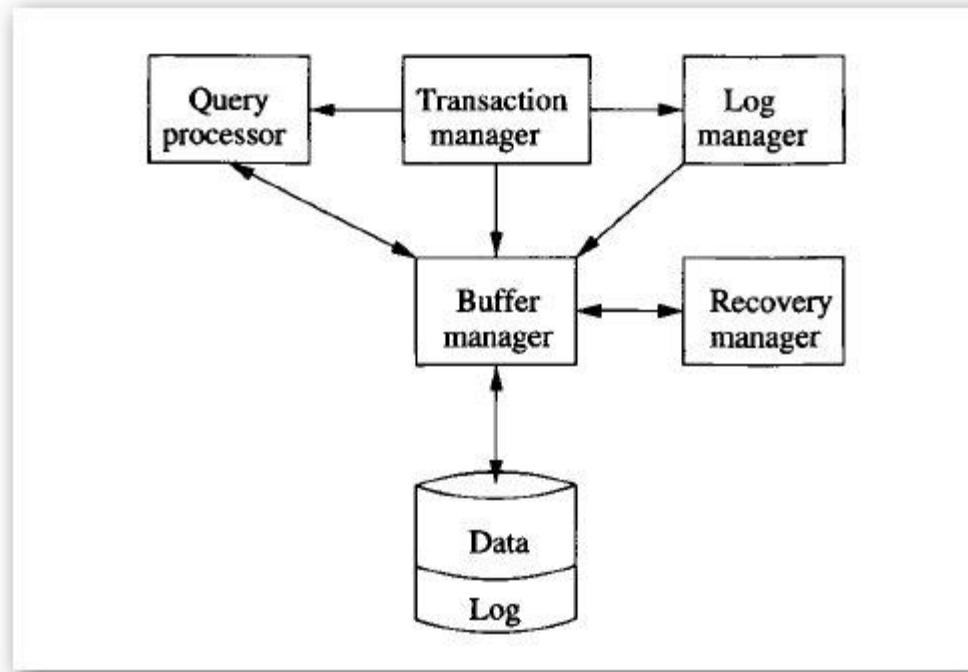
disk

Steps of a transaction and its effect on memory and disk

• <i>Action</i>	<i>t</i>	<i>M-A</i>	<i>M-B</i>	<i>D-A</i>	<i>D-B</i>
1. READ (A, t)	8	8		8	8
2. $t := t*2$	16	8		8	8
3. WRITE (A, t)	16	16		8	8
4. READ (B, t)	8	16	8	8	8
5. $t := t*2$	16	16	8	8	8
6. WRITE (B, t)	16	16	16	8	8
7. OUTPUT (A)	16	16	16	16	8
8. OUTPUT (B)	16	16	16	16	16

- Need **atomicity**: execute all actions of a transaction or none at all

One solution: undo logging
(immediate modification on disk)



The **transaction manager** will send messages about actions of transactions to the **log manager**, to the **buffer manager** about when it is possible or necessary to copy the buffer back to disk, and to the **query processor** to execute the queries and other database operations that comprise the transaction.

The **log manager** maintains the log. It must deal with the buffer manager, since space for the log initially appears in main-memory buffers, and at certain times these buffers must be copied to disk.

A log is a file of **log records**, each telling something about what some transaction has done.

Log records:

$\langle T, \text{START} \rangle$: This record indicates that transaction T has begun.

$\langle T, \text{COMMIT} \rangle$: Transaction T has completed successfully and will make no more changes to database elements.

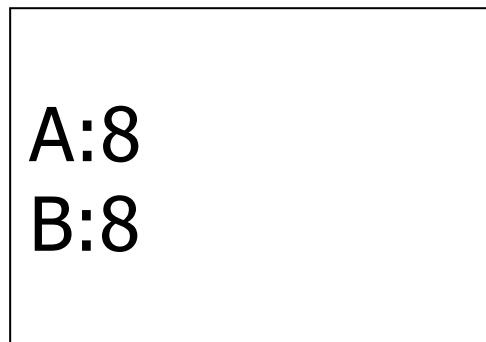
$\langle T, \text{ABORT} \rangle$: Transaction T could not complete successfully.

$\langle T, X, v \rangle$: Transaction T has changed database element X , and its former value was v .

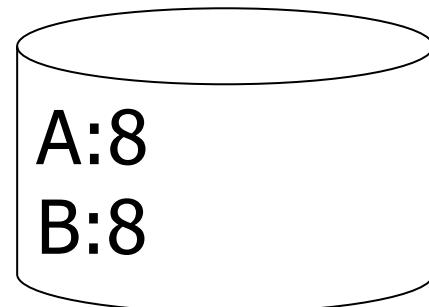
$\langle T, \text{ABORT} \rangle$ and $\langle \text{ABORT}, T \rangle$ means the same !!!

Undo logging (Immediate modification)

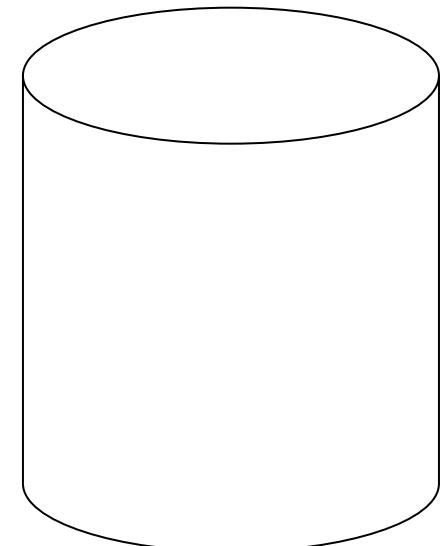
T1: Read (A,t); $t \leftarrow t \times 2$ A=B
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



memory



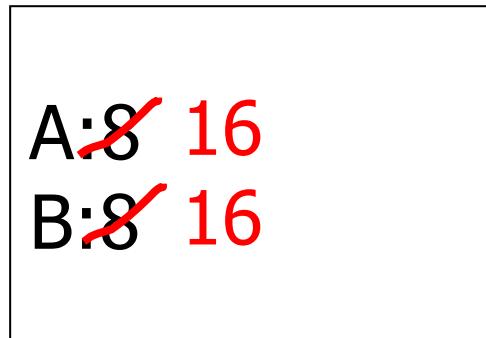
disk



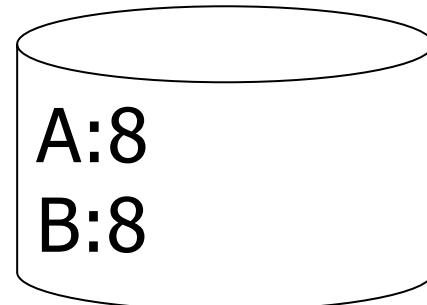
log

Undo logging (Immediate modification)

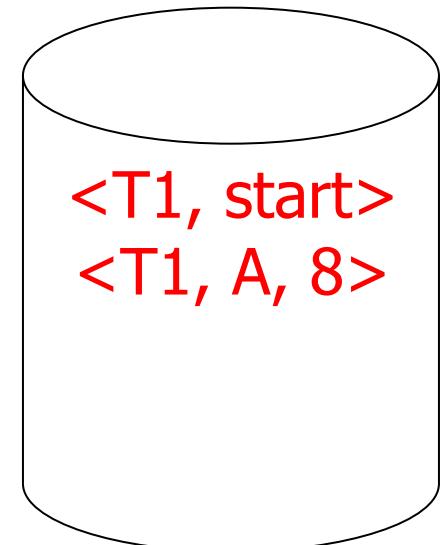
T1: Read (A,t); $t \leftarrow t \times 2$ A=B
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



memory



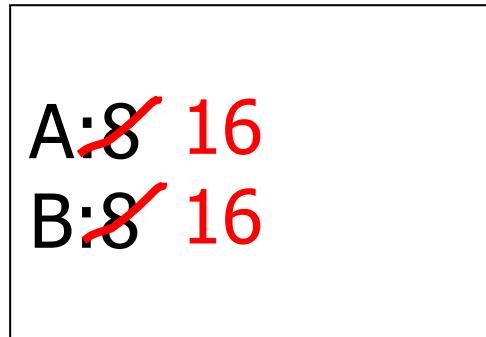
disk



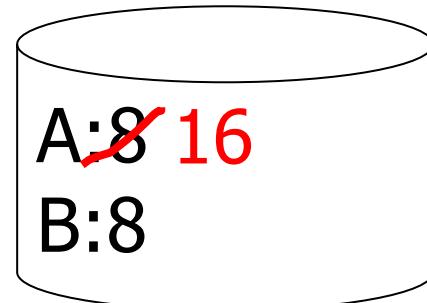
log

Undo logging (Immediate modification)

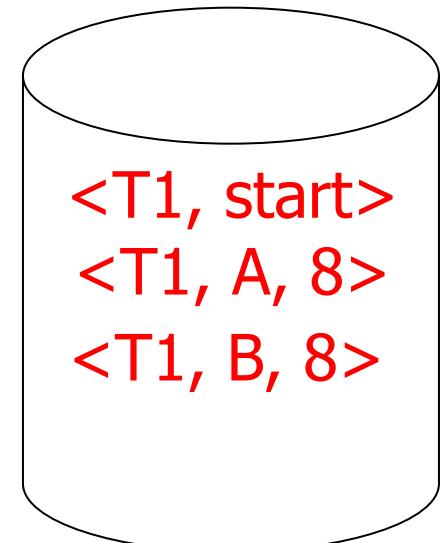
T1: Read (A,t); $t \leftarrow t \times 2$ A=B
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



memory



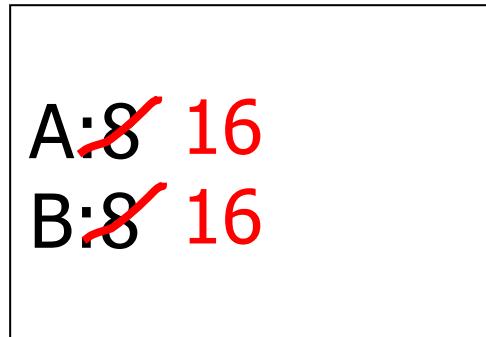
disk



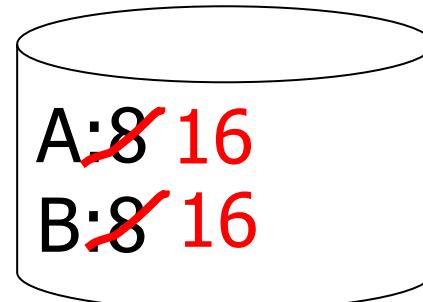
log

Undo logging (Immediate modification)

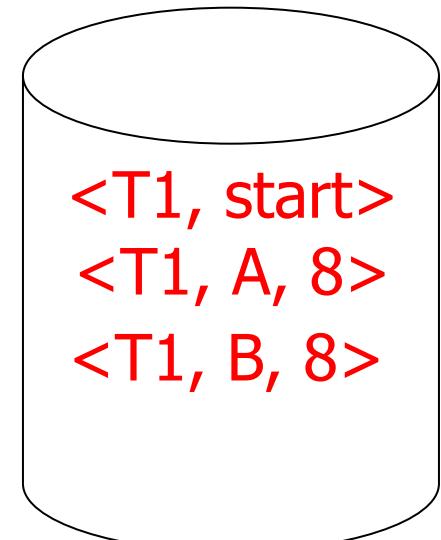
T1: Read (A,t); $t \leftarrow t \times 2$ A=B
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



memory



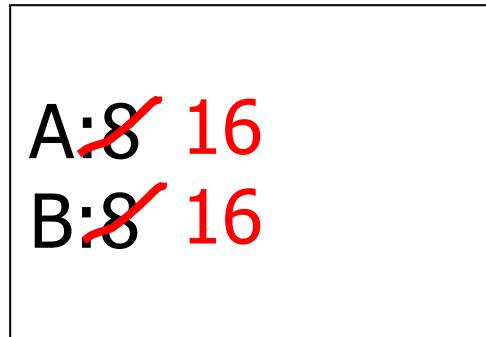
disk



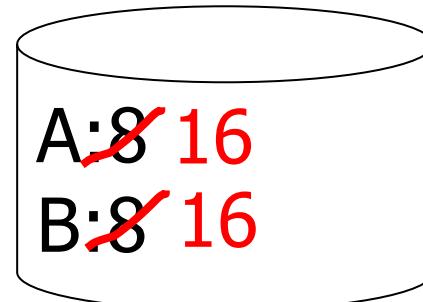
log

Undo logging (Immediate modification)

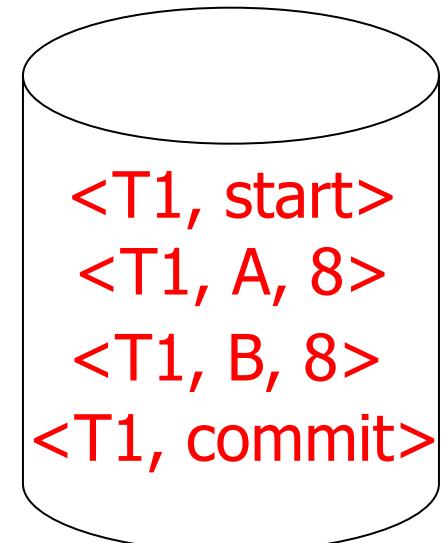
T1: Read (A,t); $t \leftarrow t \times 2$ A=B
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



memory



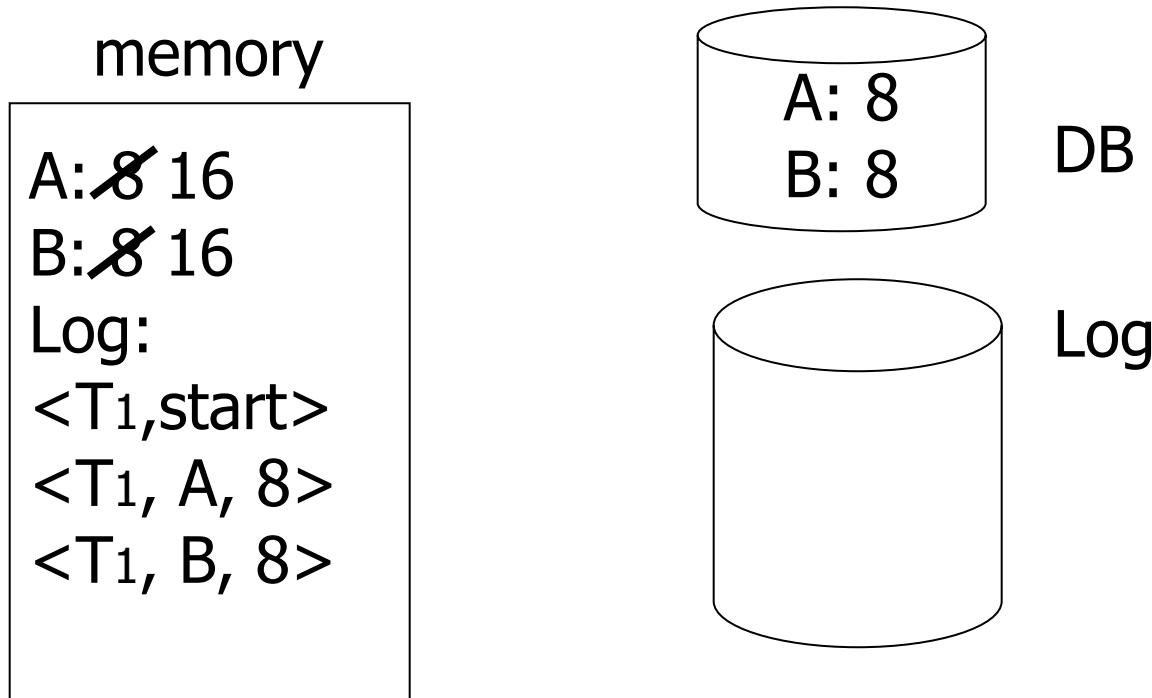
disk



log

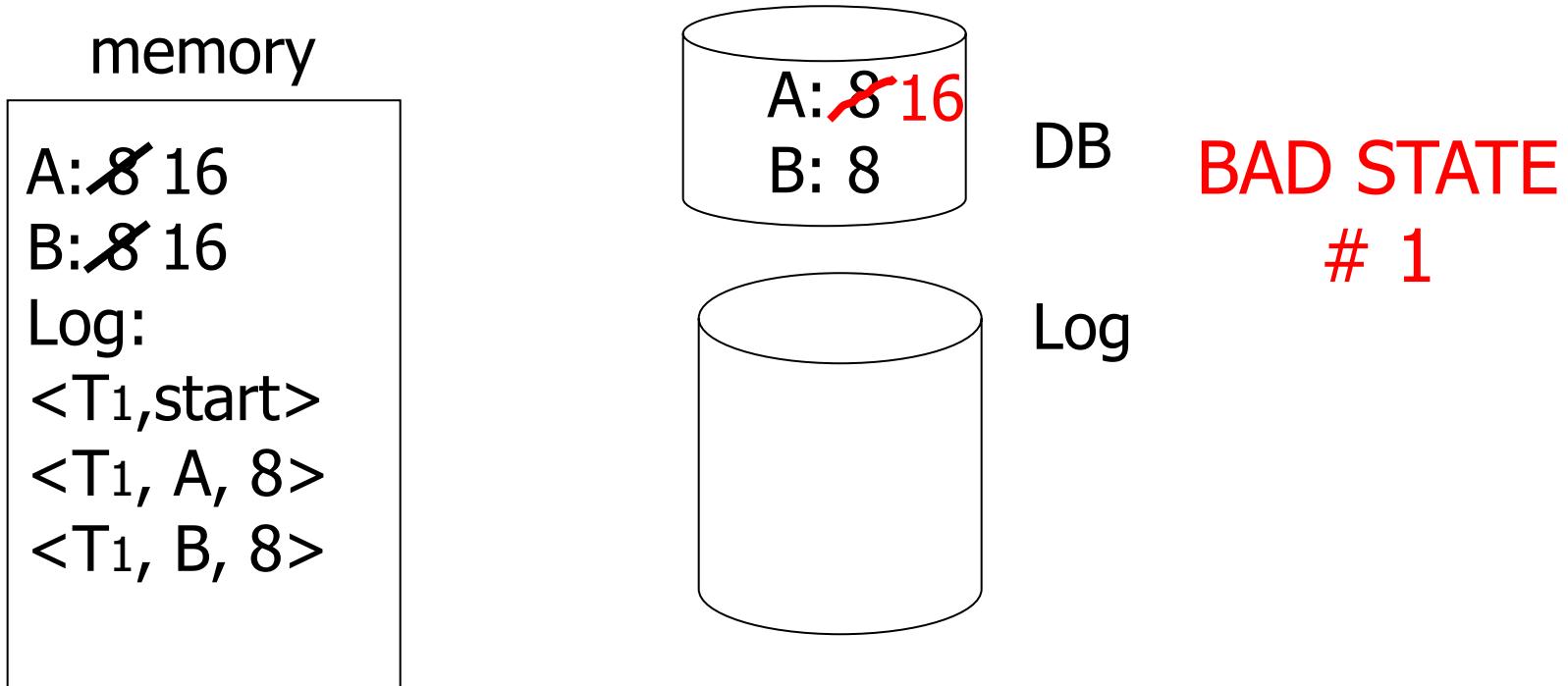
One “complication”

- Log is first written in memory
- Not written to disk on every action



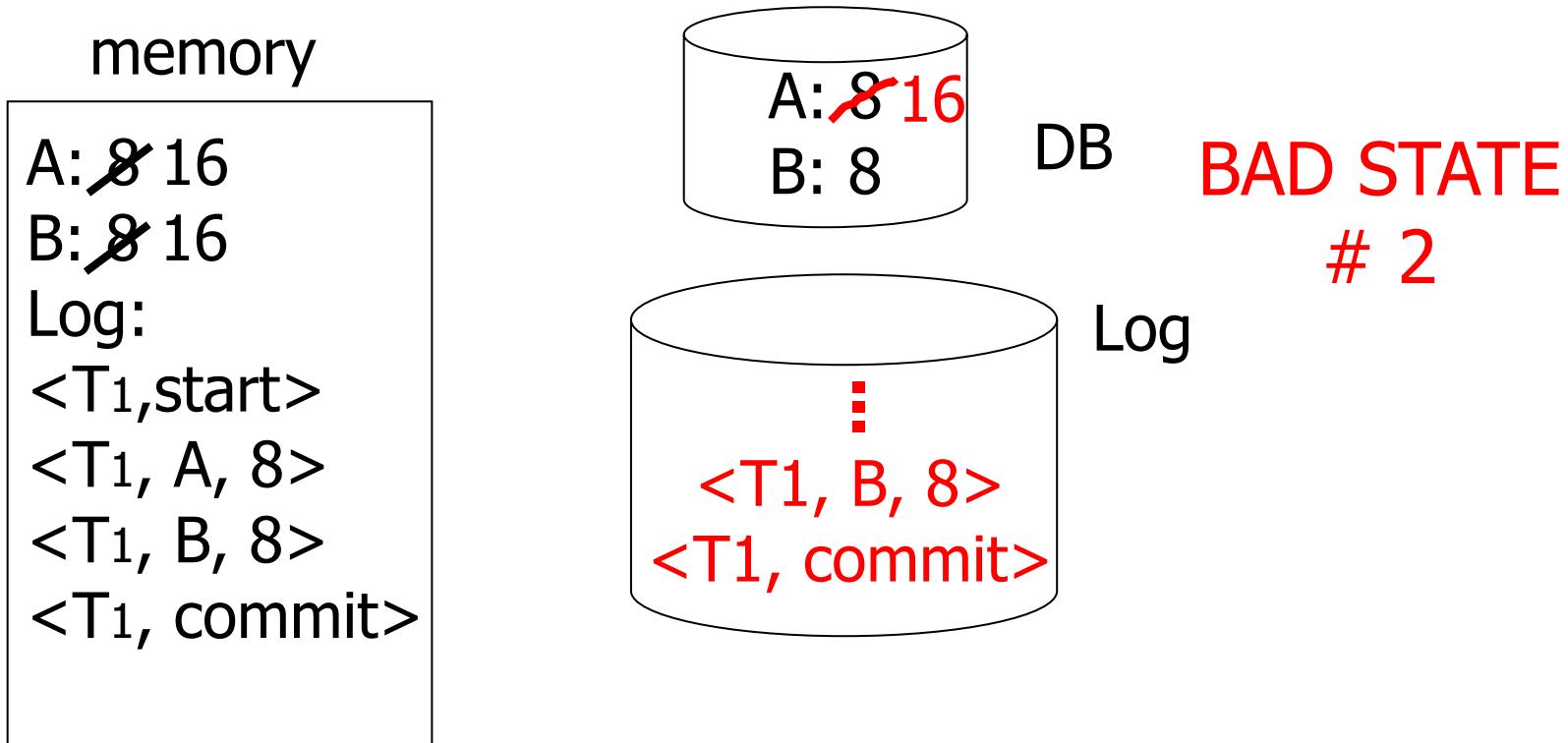
One “complication”

- Log is first written in memory
- Not written to disk on every action



One “complication”

- Log is first written in memory
- Written to disk before action



Undo logging rules

- (1) For every action generate undo log record (containing **old value**)
- (2) Before x is modified on disk, log records pertaining to x must be on disk (write ahead logging: WAL)
- (3) **Before commit** is flushed to log, **all writes** of transaction must be reflected **on disk**

Must **write** to disk in the following **order** **(UNDO LOG)**

1. The **log records** indicating changed database elements.
2. The changed **database elements** themselves.
3. The **COMMIT** log record.

Order of steps and disk writes in case of UNDO log

<i>Step</i>	<i>Activity</i>	<i>t</i>	<i>M-A</i>	<i>M-B</i>	<i>D-A</i>	<i>D-B</i>	<i>Log</i>
1)							<T , START>
2)	READ (A, t)	8	8		8	8	
3)	$t := t*2$	16	8		8	8	
4)	WRITE (A, t)	16	16		8	8	<T , A , 8>
5)	READ (B, t)	8	16	8	8	8	
6)	$t := t*2$	16	16	8	8	8	
7)	WRITE (B, t)	16	16	16	8	8	<T , B , 8>
8)	FLUSH LOG						
9)	OUTPUT (A)	16	16	16	16	8	
10)	OUTPUT (B)	16	16	16	16	16	
11)							<T , COMMIT>
12)	FLUSH LOG						

Recovery rules: Undo logging

- For every T_i with $\langle T_i, \text{start} \rangle$ in log:
 - If $\langle T_i, \text{commit} \rangle$ or $\langle T_i, \text{abort} \rangle$ in log, do nothing
 - Else $\left\{ \begin{array}{l} \text{For all } \langle T_i, X, \nu \rangle \text{ in log:} \\ \left\{ \begin{array}{l} \text{write } (X, \nu) \\ \text{output } (X) \end{array} \right. \\ \text{Write } \langle T_i, \text{abort} \rangle \text{ to log} \end{array} \right.$

Recovery rules: Undo logging

- For every T_i with $\langle T_i, \text{start} \rangle$ in log:
 - If $\langle T_i, \text{commit} \rangle$ or $\langle T_i, \text{abort} \rangle$ in log, do nothing
 - Else
 - For all $\langle T_i, X, \nu \rangle$ in log:
 - write (X, ν)
 - output (X)
 - Write $\langle T_i, \text{abort} \rangle$ to log

✗IS THIS CORRECT??

Recovery rules:

Undo logging

- (1) Let $S = \text{set of transactions with } \langle T_i, \text{start} \rangle \text{ in log, but no } \langle T_i, \text{commit} \rangle \text{ (or } \langle T_i, \text{abort} \rangle \text{) record in log}$
- (2) For each $\langle T_i, X, v \rangle$ in log,
in reverse order (latest \rightarrow earliest) do:
 - if $T_i \in S$ then $\begin{cases} - \text{write } (X, v) \\ - \text{output } (X) \end{cases}$
- (3) For each $T_i \in S$ do
 - write $\langle T_i, \text{abort} \rangle$ to log (plus **FLUSH LOG**)

What if failure during recovery?

No problem!



Undo idempotent

Recovery from Undo log

<i>Step</i>	<i>Activity</i>	<i>t</i>	<i>M-A</i>	<i>M-B</i>	<i>D-A</i>	<i>D-B</i>	<i>Log</i>
1)							<T , START>
2)	READ (A, t)	8	8		8	8	
3)	$t := t*2$	16	8		8	8	
4)	WRITE (A, t)	16	16		8	8	<T , A , 8>
5)	READ (B, t)	8	16	8	8	8	
6)	$t := t*2$	16	16	8	8	8	
7)	WRITE (B, t)	16	16	16	8	8	<T , B , 8>
8)	FLUSH LOG						
9)	OUTPUT (A)	16	16	16	16	8	
10)	OUTPUT (B)	16	16	16	16	16	
11)							<T , COMMIT>
12)	FLUSH LOG						

The crash occurs after step (12). Then the *<COMMIT T> record reached disk before the crash*. When we recover, we do not undo the results of *T*, and all log records concerning *T* are ignored by the recovery manager.

Recovery from Undo log

<i>Step</i>	<i>Activity</i>	<i>t</i>	<i>M-A</i>	<i>M-B</i>	<i>D-A</i>	<i>D-B</i>	<i>Log</i>
1)							<T , START>
2)	READ (A, t)	8	8		8	8	
3)	$t := t*2$	16	8		8	8	
4)	WRITE (A, t)	16	16		8	8	<T , A , 8>
5)	READ (B, t)	8	16	8	8	8	
6)	$t := t*2$	16	16	8	8	8	
7)	WRITE (B, t)	16	16	16	8	8	<T , B , 8>
8)	FLUSH LOG						
9)	OUTPUT (A)	16	16	16	16	8	
10)	OUTPUT (B)	16	16	16	16	16	
11)							<T , COMMIT>
12)	FLUSH LOG						

The crash occurs between steps (11) and (12). If $\langle \text{COMMIT } T \rangle$ record reached disk see previous case, if not, see next case.

Recovery from Undo log

<i>Step</i>	<i>Activity</i>	<i>t</i>	<i>M-A</i>	<i>M-B</i>	<i>D-A</i>	<i>D-B</i>	<i>Log</i>
1)							<T , START>
2)	READ (A, t)	8	8		8	8	
3)	$t := t*2$	16	8		8	8	
4)	WRITE (A, t)	16	16		8	8	<T , A , 8>
5)	READ (B, t)	8	16	8	8	8	
6)	$t := t*2$	16	16	8	8	8	
7)	WRITE (B, t)	16	16	16	8	8	<T , B , 8>
8)	FLUSH LOG						
9)	OUTPUT (A)	16	16	16	16	8	
10)	OUTPUT (B)	16	16	16	16	16	
11)							<T , COMMIT>
12)	FLUSH LOG						

The crash occurs between steps (10) and (11). Now, the COMMIT record surely was not written, so *T* is incomplete and is undone as in the previous case.

Recovery from Undo log

<i>Step</i>	<i>Activity</i>	<i>t</i>	<i>M-A</i>	<i>M-B</i>	<i>D-A</i>	<i>D-B</i>	<i>Log</i>
1)							<T , START>
2)	READ (A, t)	8	8		8	8	
3)	$t := t*2$	16	8		8	8	
4)	WRITE (A, t)	16	16		8	8	<T , A , 8>
5)	READ (B, t)	8	16	8	8	8	
6)	$t := t*2$	16	16	8	8	8	
7)	WRITE (B, t)	16	16	16	8	8	<T , B , 8>
8)	FLUSH LOG						
9)	<u>OUTPUT (A)</u>	16	16	16	16	8	
10)	OUTPUT (B)	16	16	16	16	16	
11)							<T , COMMIT>
12)	FLUSH LOG						

The crash occurs between steps (8) and (10). Again, *T is undone*. In this case the change to *A and/or B may not have reached disk*. Nevertheless, the proper value, 8, is restored for each of these database elements.

Recovery from Undo log

Step	Activity	<i>t</i>	<i>M-A</i>	<i>M-B</i>	<i>D-A</i>	<i>D-B</i>	<i>Log</i>
1)							<T , START>
2)	READ (A, <i>t</i>)	8	8		8	8	
3)	<i>t</i> := <i>t</i> *2	16	8		8	8	
4)	WRITE (A, <i>t</i>)	16	16		8	8	<T , A , 8>
5)	<u>READ (B, <i>t</i>)</u>	8	16	8	8	8	
6)	<i>t</i> := <i>t</i> *2	16	16	8	8	8	
7)	WRITE (B, <i>t</i>)	16	16	16	8	8	<T , B , 8>
8)	FLUSH LOG						
9)	OUTPUT (A)	16	16	16	16	8	
10)	OUTPUT (B)	16	16	16	16	16	
11)							<T , COMMIT>
12)	FLUSH LOG						

The crash occurs prior to step (8). Now, it is not certain whether any of the log records concerning *T* have reached disk. If the change to *A* and/or *B* reached disk, then the corresponding log record reached disk. Therefore if there were changes to *A* and/or *B* made on disk by *T*, then the corresponding log record will cause the recovery manager to undo those changes.

Checkpoint

- simple checkpoint

Periodically:

- (1) Do not accept new transactions
- (2) Wait until all transactions finish
- (3) Flush all log records to disk (log)
- (4) Flush all buffers to disk (DB) (do not discard buffers)
- (5) Write “checkpoint” record on disk (log)
- (6) Resume transaction processing

Checkpoint

- non-quiescent checkpoint

1. Write log record **<START CKPT(T1, ...Tk)>**
T1 ... Tk are active transactions, and flush log.
2. Wait until all Ti-s commit or abort, but don't prohibit other transactions from starting.
3. When all Ti-s have completed, write a log record **<END CKPT>** and flush the log.

Example: what to do at recovery?

Undo log (disk):

...	<T1,A,16>	...	<T1,commit>	Checkpoint	...	<T2,B,17>	...	<T2,commit>	...	<T3,C,21>	...
													Crash

When we scan the log backwards

If we first meet an **<END CKPT>** record, then we know that all incomplete transactions began after the previous **<START CKPT (T₁, … , T_k)>** record.

We may thus scan backwards as far as the next **<START CKPT>**, and then stop; **previous log is useless** and may as well have been discarded.

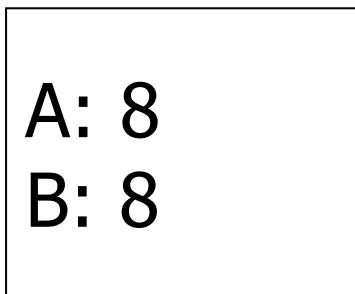
If we first meet a record **< START CKPT (T₁, … , T_k)>**, then the crash occurred during the checkpoint. We need scan no further back than the **start of the earliest of these** incomplete transactions.

To discuss:

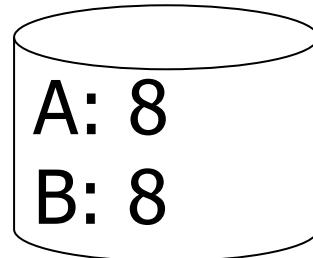
- Redo logging
- Undo/redo logging, why both?
- Real world actions
- Checkpoints
- Media failures

Redo logging (deferred modification)

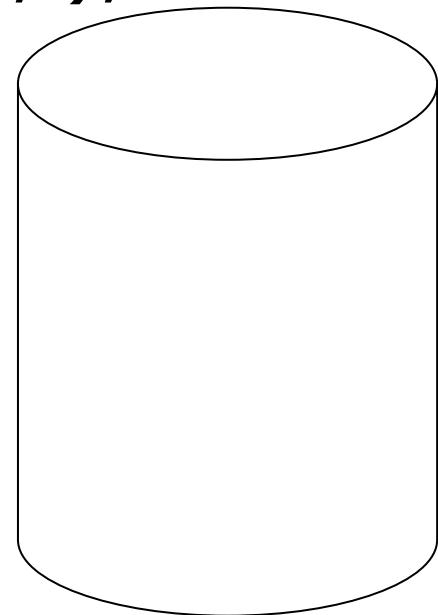
T1: Read(A,t); $t \leftarrow t \times 2$; write (A,t);
Read(B,t); $t \leftarrow t \times 2$; write (B,t);
Output(A); Output(B)



memory



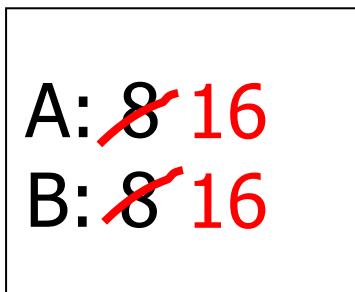
DB



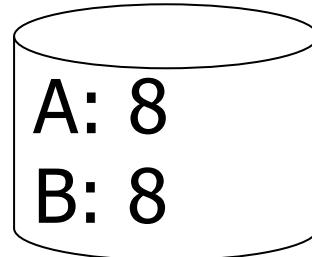
LOG

Redo logging (deferred modification)

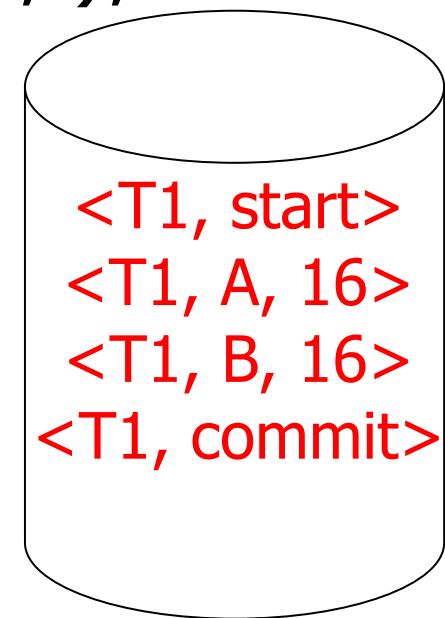
T1: Read(A,t); $t \leftarrow t \times 2$; write (A,t);
Read(B,t); $t \leftarrow t \times 2$; write (B,t);
Output(A); Output(B)



memory



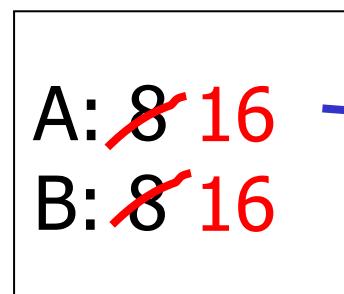
DB



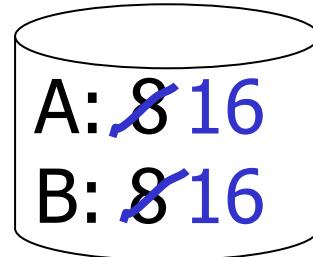
LOG

Redo logging (deferred modification)

T1: $\text{Read}(A, t); t \leftarrow t \times 2; \text{write}(A, t);$
 $\text{Read}(B, t); t \leftarrow t \times 2; \text{write}(B, t);$
 $\text{Output}(A); \text{Output}(B)$

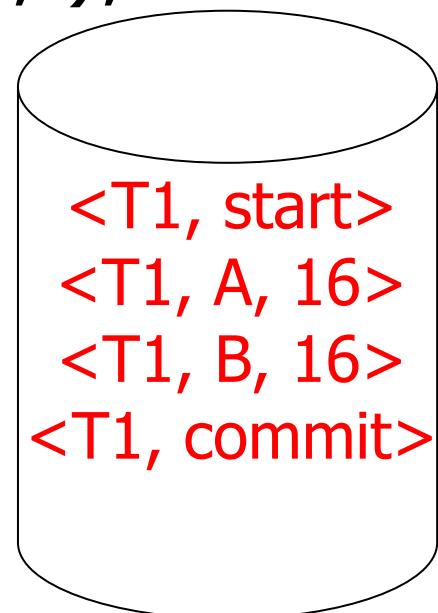


output



memory

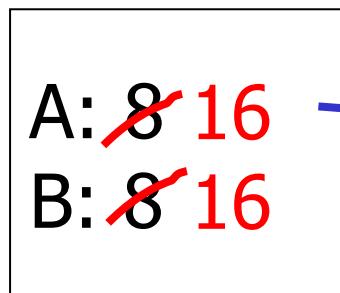
DB



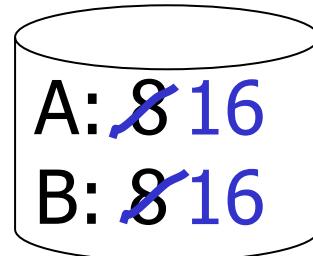
LOG

Redo logging (deferred modification)

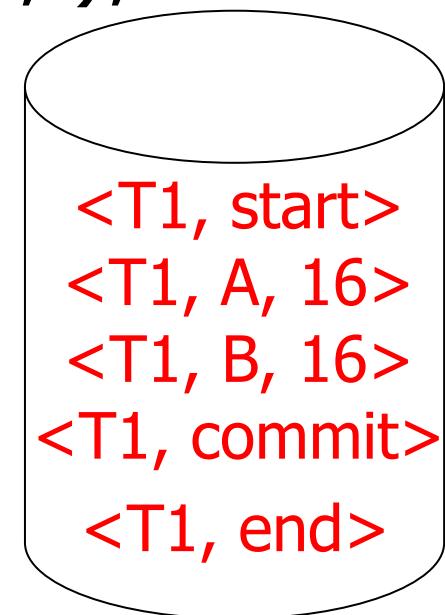
T1: $\text{Read}(A, t); t \leftarrow t \times 2; \text{write}(A, t);$
 $\text{Read}(B, t); t \leftarrow t \times 2; \text{write}(B, t);$
 $\text{Output}(A); \text{Output}(B)$



memory



DB



LOG

Redo logging rules

- (1) For every action, generate redo **log** record (containing **new value**)
- (2) Before X is modified on disk (DB), **all log records** for transaction that modified X (**including commit**) must be **on disk**
- (3) Flush log at **commit**
- (4) Write **END** record after DB updates flushed to disk

Must **write** to disk in the following **order**
(REDO LOG)

1. The **log records** indicating changed database elements.
2. The **COMMIT** log record.
3. The changed **database elements** themselves.

In Undo it was 2<->3

REDO logging rules

<i>Step</i>	<i>Activity</i>	<i>t</i>	<i>M-A</i>	<i>M-B</i>	<i>D-A</i>	<i>D-B</i>	<i>Log</i>
1)							<T , START>
2)	READ (A, t)	8	8		8	8	
3)	$t := t*2$	16	8		8	8	
4)	WRITE (A, t)	16	16		8	8	<T , A , 16>
5)	READ (B, t)	8	16	8	8	8	
6)	$t := t*2$	16	16	8	8	8	
7)	WRITE (B, t)	16	16	16	8	8	<T , B , 16>
8)							<T , COMMIT>
9)	FLUSH LOG						
10)	OUTPUT (A)	16	16	16	16	8	
11)	OUTPUT (B)	16	16	16	16	16	
12)							<T , END>
13)	FLUSH LOG						

Recovery rules:

Redo logging

- For every T_i with $\langle T_i, \text{commit} \rangle$ in log:
 - For all $\langle T_i, X, v \rangle$ in log:
 - Write(X, v)
 - Output(X)

☒ IS THIS CORRECT??

Recovery rules: Redo logging

- (1) Let $S = \text{set of transactions with } \langle T_i, \text{commit} \rangle \text{ (and no } \langle T_i, \text{end} \rangle\text{) in log}$
- (2) For each $\langle T_i, X, v \rangle$ in log, **in forward order** (earliest \rightarrow latest) do:
 - if $T_i \in S$ then $\begin{cases} \text{Write}(X, v) \\ \text{Output}(X) \end{cases}$
- (3) For each $T_i \in S$, write $\langle T_i, \text{end} \rangle$ to Log (plus **FLUSH LOG**)

Modified REDO log

We don't use $\langle T_i, end \rangle$ record for completed transactions, but use $\langle T_i, abort \rangle$ for incomplete ones.

In the Textbook you find this modified version.

Advantage: this way we use the same log records as in UNDO logging.

In practice it is best to mark both completed and incomplete transactions somehow (e.g. $\langle T, end \rangle$, $\langle T, abort \rangle$).

Recovery rules: Modified Redo log

- (1) Let $S = \text{set of transactions with } \langle T_i, \text{commit} \rangle \text{ in log}$
- (2) For each $\langle T_i, X, v \rangle$ in log, in **forward order** (earliest \rightarrow latest) do:
 - if $T_i \in S$ then $\begin{cases} \text{Write}(X, v) \\ \text{Output}(X) \end{cases}$
- (3) For each T_i NOT in S , write $\langle T_i, \text{abort} \rangle$ to Log (plus **FLUSH LOG**)

Checkpoint

- non-quiescent checkpoint

1. Write log record **<START CKPT(T1, ...Tk)>**

T1 ... Tk are active (uncommitted) transactions, and flush log.

2. Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already **committed** when the **<START CKPT>** record was written to the log. (**dirty buffers**)

3. Write an **<END CKPT>** record to the log and flush the log.

Example: what to do at recovery?

Redo log (disk):

...	<T1,A,16>	...	<T1,commit>	Checkpoint	...	<T2,B,17>	...	<T2,commit>	...	<T3,C,21>	...
													Crash

When we scan the log backwards

<START T1>
<T1, A, 5>
<START T2>
<COMMIT T1>
<T2, B, 10>
<START CKPT (T2)>
<T2, C, 15>
<START T3>
<T3, D, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

If we first meet an **<END CKPT>** record, we need to scan no further back than the earliest of **<START Ti>** records (among corresponding **<START CKPT(T1, T2, ... Tk)>**).

If we first meet a record **< START CKPT (T1, ... , Tk)>**, then the crash occurred during the checkpoint. We cannot be sure that committed transactions prior to the start of this checkpoint had their changes written to disk. Thus, we must search back to the previous **<END CKPT>** record, find its matching **< START CKPT (T1, ... , Tk)>**.

When we scan the log backwards

<START CKPT (...)>

...

<END CKPT>

<START T1>

<T1, A, 5>

<START T2>

<COMMIT T1>

<T2, B, 10>

<START CKPT (T2)>

<T2, C, 15>

<START T3>

<T3, D, 20>

If we first meet an <END CKPT> record, we need to scan no further back than the earliest of <START Ti> records (among corresponding <START CKPT(T1, T2, ... Tk)>).

If we first meet a record

< START CKPT (T1, ... , Tk)>, then the crash occurred during the checkpoint.

We cannot be sure that committed transactions prior to the start of this checkpoint had their changes written to disk.

Thus, we must search back to the previous <END CKPT> record, find its matching < START CKPT (T1, ... , Tk)>.

Key drawbacks:

- *Undo logging*: need frequent disk writes
- *Redo logging*: need to keep all modified blocks in memory until commit

Solution: **undo/redo** logging!

Update \Rightarrow $\langle T_i, X, \text{Old } X \text{ val, New } X \text{ val} \rangle$

Rules

- Page X can be flushed **before** or
after T_i commit
- Log record flushed before corresponding updated page (WAL)
- Flush at commit (log only)

UR1 *Before modifying any database element X on disk because of changes made by some transaction T , it is necessary that the update record $\langle T, X, v, w \rangle$ appear on disk.*

UR2 *A $\langle \text{COMMIT } T \rangle$ record must be flushed to disk as soon as it appears in the log.*

Must **write** to disk in the following **order**
(UNDO/REDO LOG)

1. The **log records** indicating changed database elements.
 - The **COMMIT** log record.
 - The changed **database elements** themselves.

Last 2 can be in any order!

Example: Undo/Redo logging what to do at recovery?

log (disk):

...	<checkpoint>	...	<T1, A, 10, 15>	...	<T1, B, 20, 23>	...	<T1, commit>	...	<T2, C, 30, 38>	...	<T2, D, 40, 41>	...	Crash
-----	--------------	-----	-----------------	-----	-----------------	-----	--------------	-----	-----------------	-----	-----------------	-----	-------

The **undo/redo recovery** policy is:

1. **Redo** all the **committed** transactions in the order **earliest-first**, and
2. **Undo** all the **incomplete** transactions in the order **latest-first**.

Checkpoint

- non-quiescent checkpoint

1. Write log record <START CKPT(T1, ...Tk)>

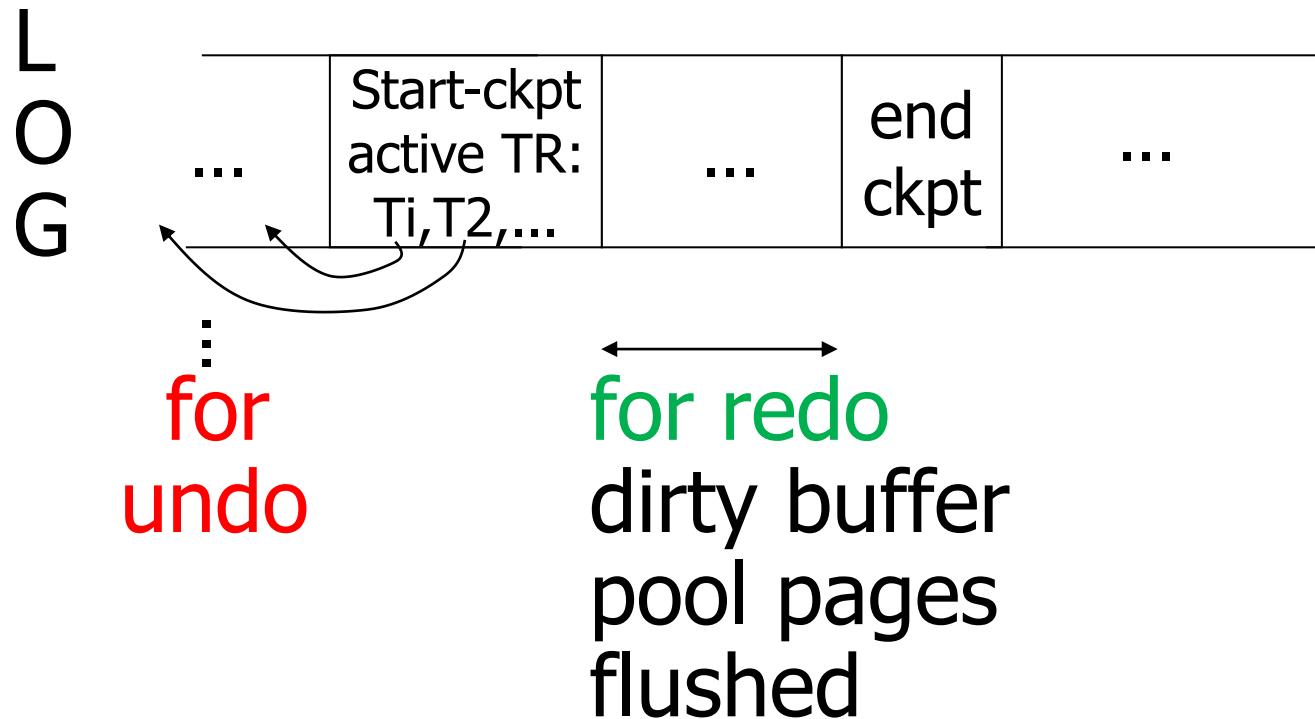
T1 ... Tk are active (uncommitted) transactions, and flush log.

2. Write to disk all the buffers that are *dirty*; i.e., they contain one or more changed database elements.

Unlike redo logging, we **flush all dirty buffers**, not just those written by committed transactions.

3. Write a log record <END CKPT> and flush the log.

Non-quiescent checkpoint



When we scan the log backwards

```
<START T1>  
<T1, A, 4, 5>  
<START T2>  
<COMMIT T1>  
<T2, B, 9, 10>  
<START CKPT (T2)>  
<T2, C, 14, 15>  
<START T3>  
<T3, D, 19, 20>  
<END CKPT>  
<COMMIT T2>  
<COMMIT T3>
```

If we first meet an **<END CKPT>** record:

When we **redo** a transaction, we do not need to look prior to the **<START CKPT>** record, because we know that changes prior to the start of the checkpoint were flushed to disk during the checkpoint.
(e.g. T2)

If we first meet a **<START CKPT>** record:
We simply ignore it, find **<END CKPT>**.

When we scan the log backwards

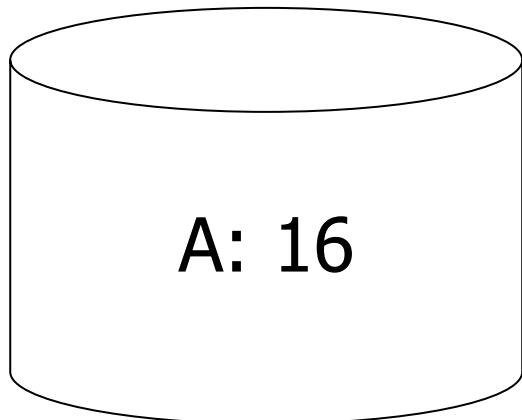
```
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>
<T2, B, 9, 10>
<START CKPT (T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<END CKPT>
<COMMIT T3>
```

If we first meet an <END CKPT> record:

However when we undo a transaction, we have to look prior to the <START-CKPT> record to find if there were actions that may have reached disk and need to be undone. (e.g. T2)

If we first meet a <START CKPT> record: We simply ignore it, find <END CKPT>.

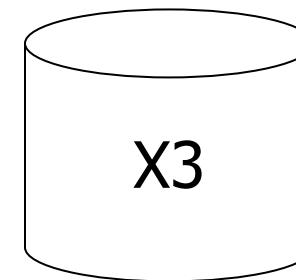
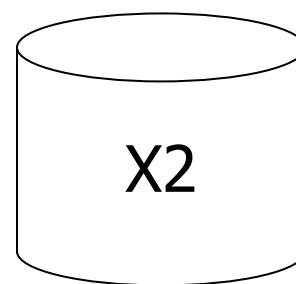
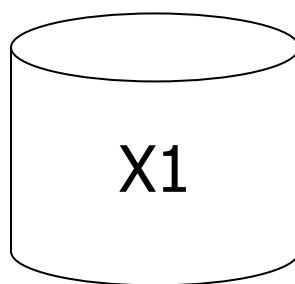
Media failure (loss of non-volatile storage)



Solution: Make copies of data!

Example 1 Triple modular redundancy

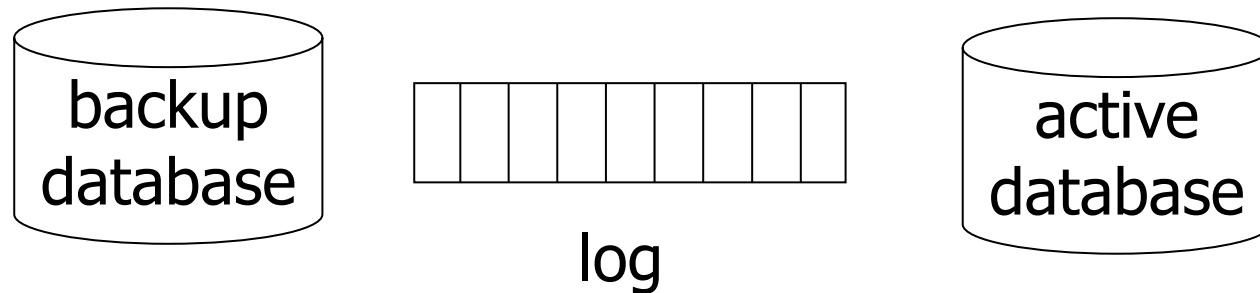
- Keep 3 copies on separate disks
- Output(X) \rightarrow three outputs
- Input(X) \rightarrow three inputs + vote (if different)



Example #2 Redundant writes, Single reads

- **Keep N copies** on separate disks
 - Output(X) \rightarrow N outputs
 - Input(X) \rightarrow Input one copy
 - if ok, done
 - else try another one
- \Leftrightarrow Assumes bad data can be detected

Example #3: DB Dump + Log



- If active database is lost,
 - restore active database from backup
 - bring up-to-date **using redo entries in log**

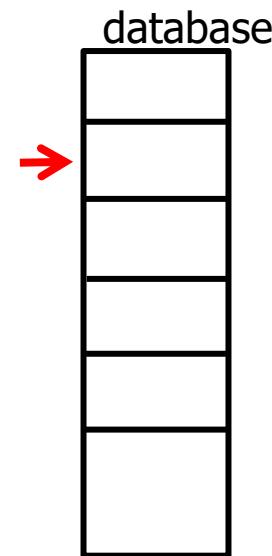
Backup Database

- Just like checkpoint,
except that we write full database

create backup database:

```
for i := 1 to DB_Size do
    [read DB block i; write to backup]
```

[transactions run concurrently]



Backup Database

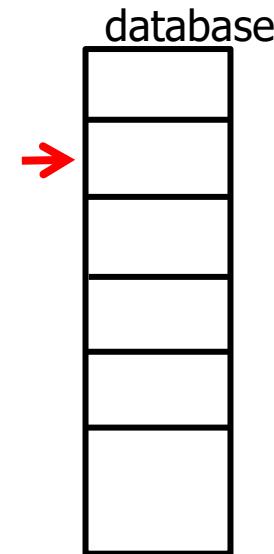
- Just like checkpoint,
except that we write full database

create backup database:

for i := 1 to DB_Size do

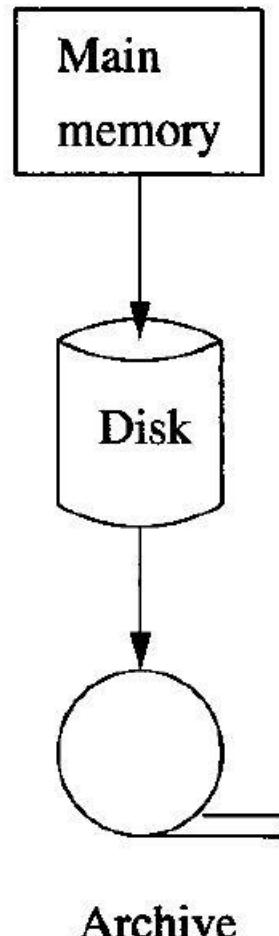
[read DB block i; write to backup]

[transactions run concurrently]



- **Restore** from backup DB and log:
Similar to recovery from checkpoint and log

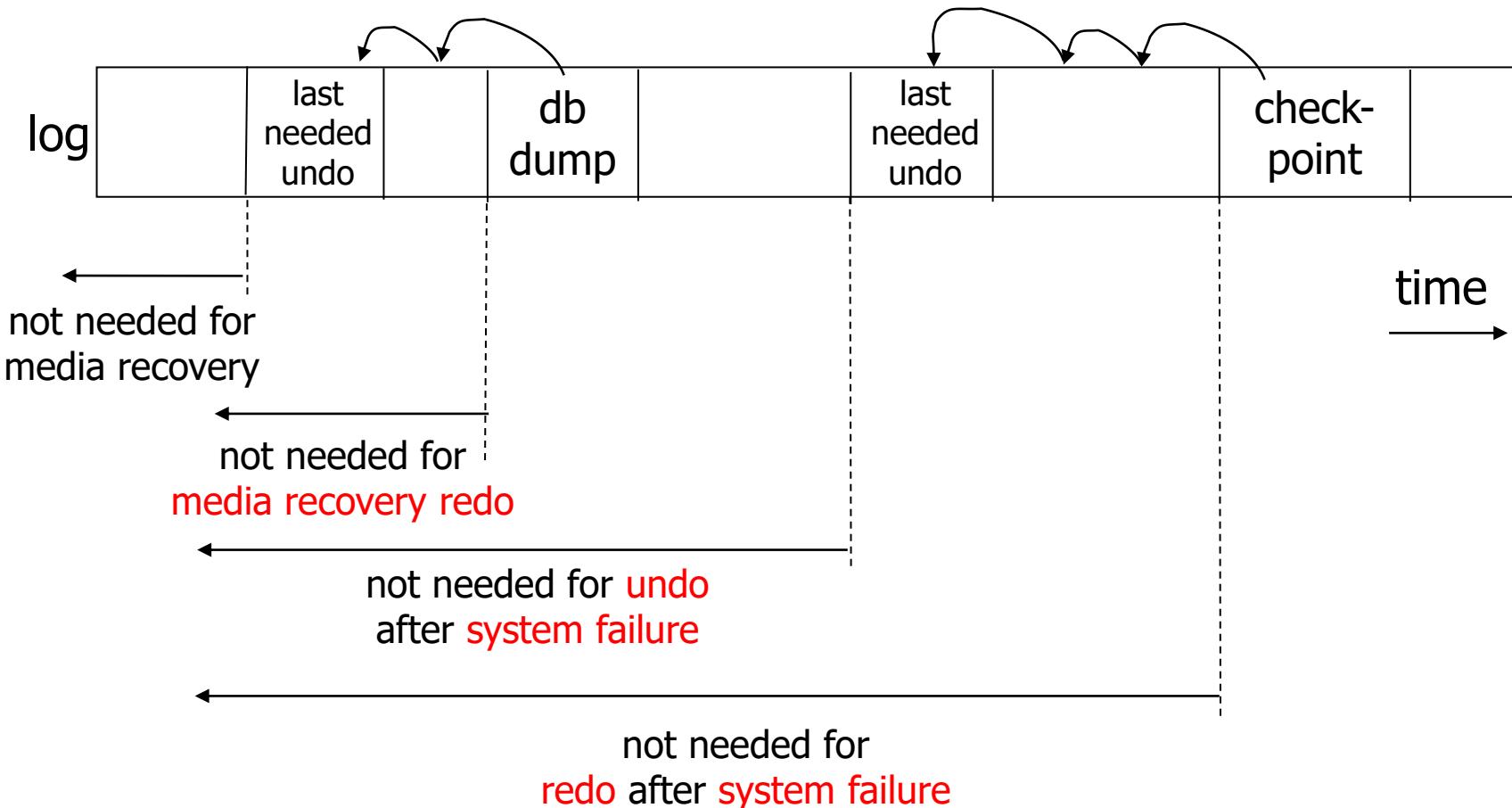
Restore from backup (archive) and log



Checkpoint gets data from memory to disk;
log allows recovery from system failure

Dump gets data from disk to archive;
archive plus log allows recovery from media failure

When can log be discarded?



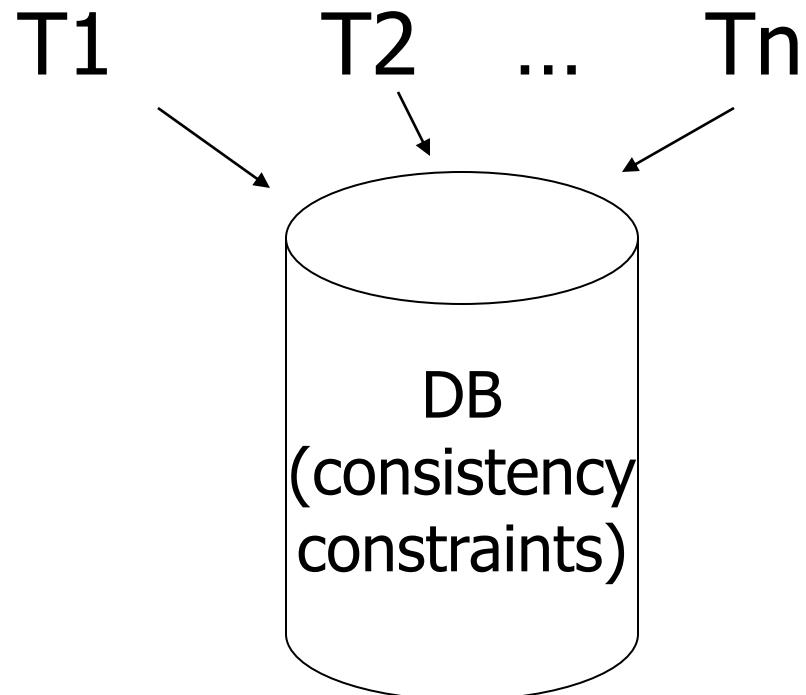
Summary

- Consistency of data
- One source of problems: failures
 - Logging
 - Redundancy
- Another source of problems:
Data Sharing..... next

Ullman et al. : Database System Principles

Textbook ch. 18: **Concurrency Control**

Chapter 18 [18] Concurrency Control



Interactions among concurrently executing transactions can cause the database state to become inconsistent, even when the transactions individually preserve correctness of the state, and there is no system failure.

Thus, the timing of individual steps of different transactions needs to be regulated in some manner.

This regulation is the job of the *scheduler component of the DBMS*, and the general process of assuring that transactions preserve consistency when executing simultaneously is called *concurrency control*.

In most situations, the **scheduler** will execute the **reads and writes directly**, first calling on the buffer manager if the desired database element is not in a buffer.

However, in some situations, it is not safe for the request to be executed immediately.

The **scheduler must delay the request**.

In some concurrency-control techniques, the **scheduler may even abort the transaction** that issued the request

Example:

T1: Read(A)

$A \leftarrow A + 100$

Write(A)

Read(B)

$B \leftarrow B + 100$

Write(B)

T2: Read(A)

$A \leftarrow A \times 2$

Write(A)

Read(B)

$B \leftarrow B \times 2$

Write(B)

Constraint: $A=B$

A *schedule* is a sequence of the important actions taken by one or more transactions.

When studying concurrency control, the important read and write actions take place in the main-memory buffers, not the disk.

That is, a database element A that is brought to a buffer by some transaction T may be read or written in that buffer not only by T but by other transactions that access A .

A schedule is *serial* if its actions consist of all the actions of one transaction, then all the actions of another transaction, and so on.

No mixing of the actions is allowed.

Schedule A (serial)

T1

Read(A); $A \leftarrow A + 100$

Write(A);

Read(B); $B \leftarrow B + 100$;

Write(B);

T2

Read(A); $A \leftarrow A \times 2$;
Write(A);
Read(B); $B \leftarrow B \times 2$;
Write(B);

A	B
25	25
125	
125	
250	
250	
250	250

Schedule B (serial)

T1

T2

Read(A); A \leftarrow A+100
Write(A);
Read(B); B \leftarrow B+100;
Write(B);

Read(A); A \leftarrow A \times 2;
Write(A);
Read(B); B \leftarrow B \times 2;
Write(B);

A	B
25	25
	50
	50
	150
150	150

A schedule S is *serializable* if there is a serial schedule S' such that for every initial database state, the *effects* of S and S' are the *same*.

Schedule C (Serializable)

T1

Read(A); $A \leftarrow A + 100$

Write(A);

Read(B); $B \leftarrow B + 100$;
Write(B);

T2

Read(A); $A \leftarrow A \times 2$;
Write(A);

Read(B); $B \leftarrow B \times 2$;
Write(B);

A	B
25	25
125	
250	
125	
250	
250	250

For any initial values!

Schedule D (not serializable)

T1

Read(A); $A \leftarrow A + 100$

Write(A);

Read(B); $B \leftarrow B + 100$;

Write(B);

T2

Read(A); $A \leftarrow A \times 2$;
Write(A);
Read(B); $B \leftarrow B \times 2$;
Write(B);

A	B
25	25
	125
	250
	50
	150
250	150

Schedule E

Same as Schedule D
but with new T2'

T1

Read(A); $A \leftarrow A + 100$

Write(A);

Read(B); $B \leftarrow B + 100$;

Write(B);

T2'

Read(A); $A \leftarrow A + 1$;

Write(A);

Read(B); $B \leftarrow B + 1$;

Write(B);

A	B
25	25
125	
126	
26	
126	
126	126

Is it serializable ???

- Want schedules that are “good”, regardless of
 - initial state and
 - transaction semantics
- Only look at **order of read and writes**

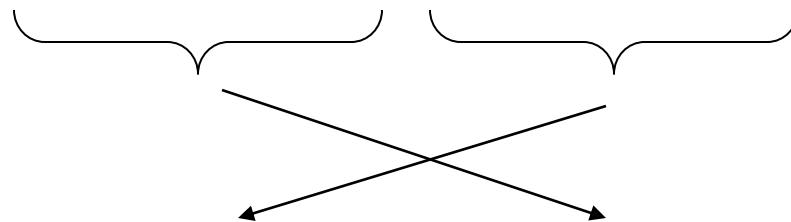
It is not realistic for the scheduler to concern itself with the details of computation undertaken by transactions.

Any database element A *that a transaction T writes is given a value* that depends on the database state in such a way that **no arithmetic coincidences occur**.

To make the notation precise:

1. An *action* is an expression of the form $r_i(X)$ or $W_i(X)$, meaning that transaction T_i , *reads* or *writes*, respectively, the database element X .
2. A *transaction* T_i is a sequence of actions with subscript i .
3. A *schedule* S of a set of transactions T is a sequence of actions, in which for each transaction T_i in T , the *actions of T_i appear in S in the same order* that they appear in the definition of T_i itself. We say that S is an interleaving of the actions of the transactions of which it is composed.

Example:

$$Sc = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$$

$$Sc' = r_1(A)w_1(A) \ r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$$


T_1

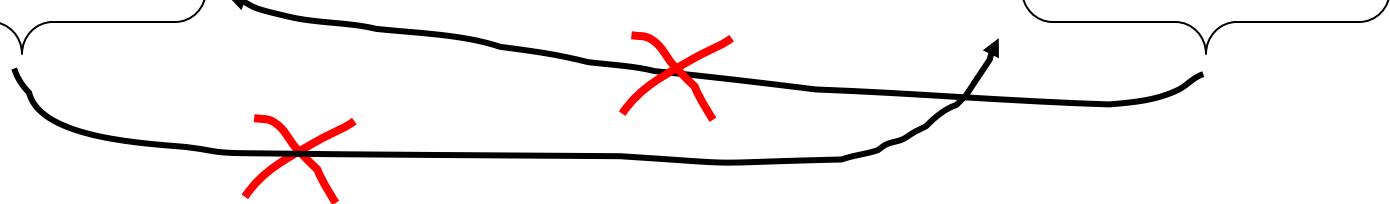
T_2

Sc is "equivalent" to a serial schedule.

Sc is "good" (every data element is read and written first by T1)

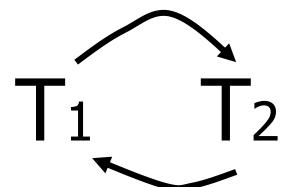
However, for S_d :

$$S_d = r_1(A)w_1(A)r_2(A)w_2(A) \quad r_2(B)w_2(B)r_1(B)w_1(B)$$



- as a matter of fact,
 T_2 must precede T_1
in any equivalent schedule,
i.e., $T_2 \rightarrow T_1$

- $T_2 \rightarrow T_1$
- Also, $T_1 \rightarrow T_2$



- S_d cannot be rearranged into a serial schedule
- S_d is not “equivalent” to any serial schedule
- S_d is “bad”

Returning to Sc

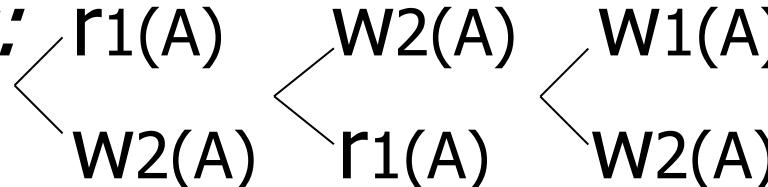
$$Sc = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$$

$$T_1 \rightarrow T_2$$
$$T_1 \rightarrow T_2$$

- ☞ **no cycles** \Rightarrow Sc is “equivalent” to a serial schedule (in this case T_1, T_2)

Concepts

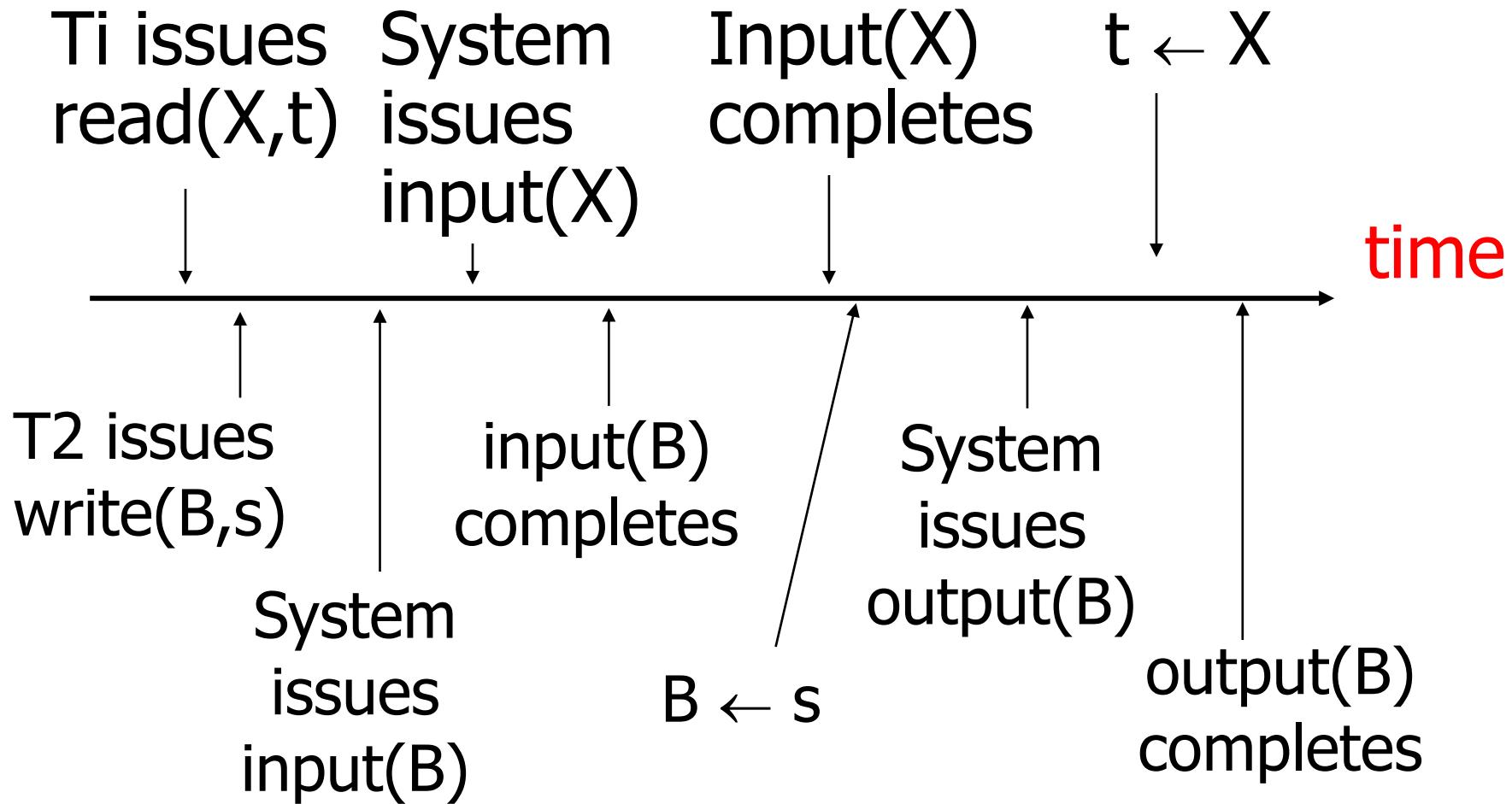
Transaction: sequence of $r_i(x)$, $w_i(x)$ actions

Conflicting actions: 

Schedule: represents chronological order
in which actions are executed

Serial schedule: no interleaving of actions
or transactions

What about concurrent actions?

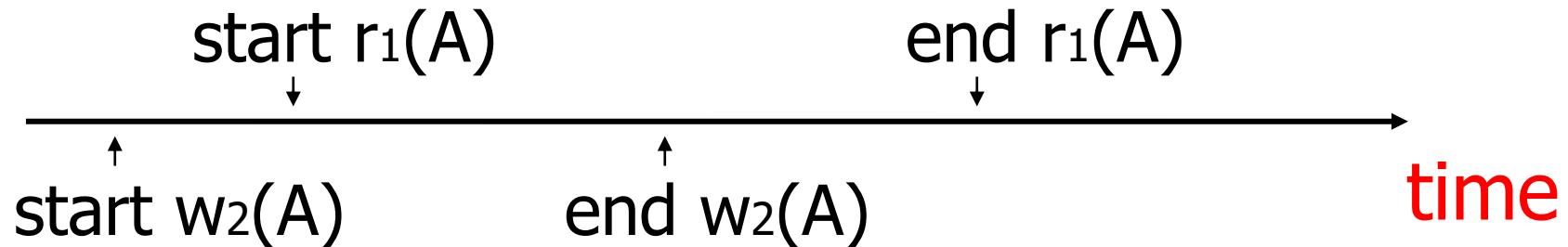


So net effect is either

- $S = \dots r_1(X) \dots w_2(B) \dots$ or
- $S = \dots w_2(B) \dots r_1(X) \dots$

We **assume** that elementary **actions** are **atomic**, and follow each other.

What about conflicting, concurrent actions on same object?



- **Assume** equivalent to either $r_1(A)$ $w_2(A)$
or $w_2(A)$ $r_1(A)$
- \Rightarrow low level synchronization mechanism
- Assumption called “atomic actions”

Definition

S_1, S_2 are conflict equivalent schedules

if S_1 can be transformed into S_2 by a series of non-conflicting swaps of adjacent actions.

A schedule is conflict serializable if it is conflict equivalent to some serial schedule.

Example:

$r_1(A); w_1(A); r_2(A); \underline{w_2(A)}; \underline{r_1(B)}; w_1(B); r_2(B); w_2(B);$

We claim this schedule is conflict-serializable.

1. $r_1(A); w_1(A); r_2(A); \underline{w_2(A)}; \underline{r_1(B)}; w_1(B); r_2(B); w_2(B);$
2. $r_1(A); w_1(A); \underline{r_2(A)}; \underline{r_1(B)}; w_2(A); w_1(B); r_2(B); w_2(B);$
3. $r_1(A); w_1(A); r_1(B); r_2(A); \underline{w_2(A)}; \underline{w_1(B)}; r_2(B); w_2(B);$
4. $r_1(A); w_1(A); r_1(B); \underline{r_2(A)}; \underline{w_1(B)}; w_2(A); r_2(B); w_2(B);$
5. $r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B);$

- Conflict-serializability is a **sufficient** condition for serializability i.e., a conflict-serializable schedule is a serializable schedule.
- Conflict-serializability is **not required** for a schedule to be serializable.
- Schedulers **in commercial systems** generally use conflict-serializability when they need to guarantee serializability.

Precedence graph $P(S)$ (S is schedule)

Nodes: transactions in S ($T_1, T_2 \dots$)

Arcs: $T_i \rightarrow T_j$ whenever

- $p_i(A), q_j(A)$ are actions in S
- $p_i(A) <_S q_j(A)$ ($p_i(A)$ precedes $q_j(A)$ in S)
- at least one of p_i, q_j is a write

Exercise:

- What is $P(S)$ for

$S = w_3(A) \ w_2(C) \ r_1(A) \ w_1(B) \ r_1(C) \ w_2(A) \ r_4(A) \ w_4(D)$

- Is S serializable?

Another Exercise:

- What is $P(S)$ for
 $S = w_1(A) \ r_2(A) \ r_3(A) \ w_4(A)$?

Lemma

S_1, S_2 conflict equivalent $\Rightarrow P(S_1) = P(S_2)$

Proof:

Assume $P(S_1) \neq P(S_2)$

$\Rightarrow \exists T_i: T_i \rightarrow T_j$ in S_1 and not in S_2

$\Rightarrow S_1 = \dots p_i(A) \dots q_j(A) \dots$ $\left. \begin{array}{l} p_i, q_j \\ \text{conflict} \end{array} \right\}$

$S_2 = \dots q_j(A) \dots p_i(A) \dots$

$\Rightarrow S_1, S_2$ not conflict equivalent

Note: $P(S_1)=P(S_2) \not\Rightarrow S_1, S_2$ conflict equivalent

Counter example:

$S_1=w_1(A) \ r_2(A) \quad w_2(B) \ r_1(B)$ (cannot swap)

$S_2=r_2(A) \ w_1(A) \quad r_1(B) \ w_2(B)$

Theorem

$P(S_1)$ acyclic $\iff S_1$ conflict serializable

(\Leftarrow) Assume S_1 is conflict serializable

$\Rightarrow \exists S_s: S_s, S_1$ conflict equivalent

$\Rightarrow P(S_s) = P(S_1)$

$\Rightarrow P(S_1)$ acyclic since $P(S_s)$ is acyclic

Theorem

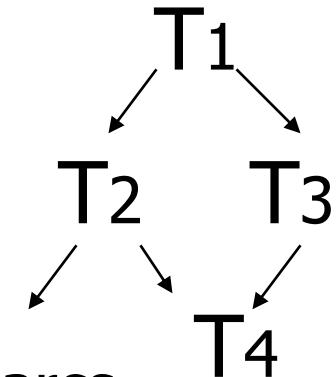
$P(S_1)$ acyclic $\iff S_1$ conflict serializable

(\Rightarrow) Assume $P(S_1)$ is acyclic

Transform S_1 as follows:

- (1) Take T_1 to be transaction with no incident arcs
- (2) **Move all T_1 actions to the front**

$S_1 = \dots \dots q_j(A) \dots \dots p_1(A) \dots \dots$



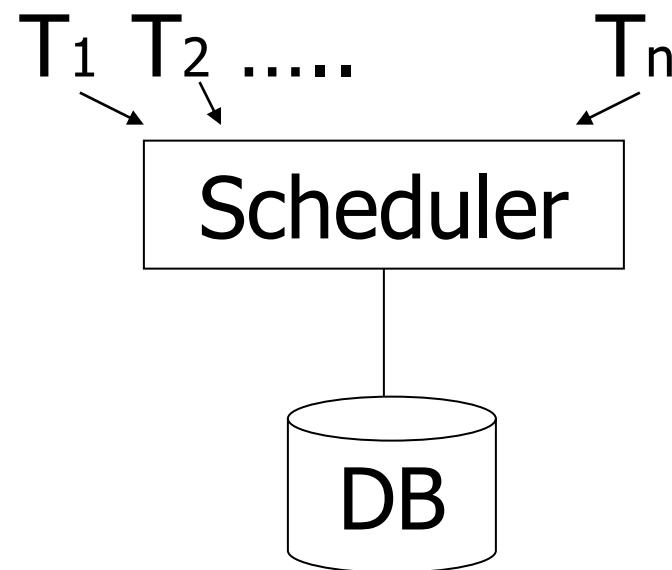
- (3) we now have $S_1 = < T_1 \text{ actions} > < \dots \text{ rest} \dots >$
- (4) **repeat above steps** to serialize rest!

How to enforce serializable schedules?

Option 1: run system, recording $P(S)$;
at end of day, **check for** $P(S)$
cycles and declare if execution
was good

How to enforce serializable schedules?

Option 2: prevent P(S) cycles from occurring

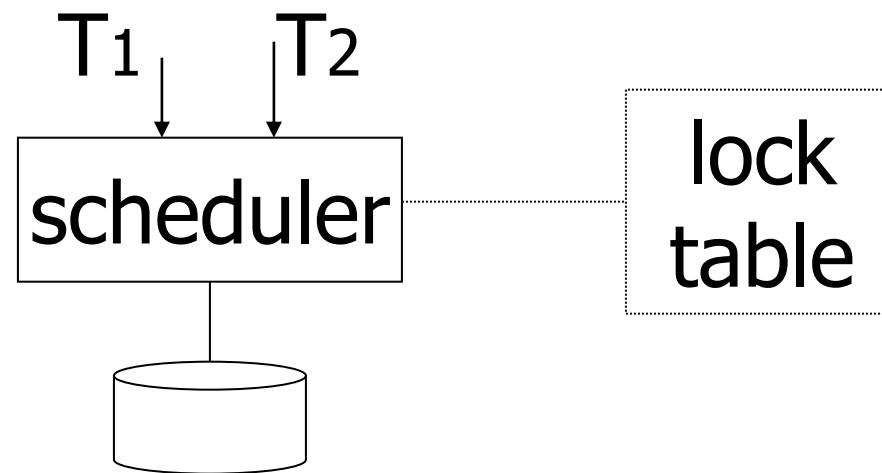


A locking protocol

Two new actions:

lock (exclusive): $l_i(A)$

unlock: $u_i(A)$



Rule #1: Consistency of transactions

$T_i: \dots l_i(A) \dots p_i(A) \dots u_i(A) \dots$

1. A transaction can only read or write an element if it **previously was granted a lock** on that element and hasn't yet released the lock.
2. If a transaction locks an element, it must **later unlock** that element.

Rule #2

Legality of schedules

$$S = \dots \text{li}(A) \dots \text{ui}(A) \dots$$
$$\longleftrightarrow$$
$$\text{no } \text{lj}(A)$$

Locks must have their intended meaning: no two transactions may have locked the same element without one having first released the lock.

Exercise:

- What **schedules** are **legal**?
What **transactions** are **consistent**?

$$S1 = l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B) \\ r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$$
$$S2 = l_1(A)r_1(A)w_1(B)u_1(A)u_1(B) \\ l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$$
$$S3 = l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B) \\ l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$$

Exercise:

- What schedules are legal?
What transactions are consistent?

$S1 = l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)$
 $r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$ (S1: not legal)

$S2 = l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)$ (T1: not consistent)
 $l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$ (S2: not legal)

$S3 = l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)$
 $l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$
(S3: legal schedule, T1,T2,T3: consistent transactions)

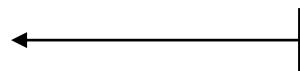
Schedule F (legal schedule of consistent transactions)

(still not equivalent to a serial schedule)

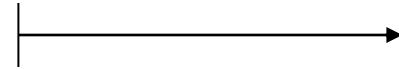
	A	B
T1	25	25
l ₁ (A);Read(A)		
A \leftarrow A+100;Write(A);u ₁ (A)	125	
T2		
l ₂ (A);Read(A)	250	
A \leftarrow Ax2;Write(A);u ₂ (A)		
l ₂ (B);Read(B)		
B \leftarrow Bx2;Write(B);u ₂ (B)	50	
l ₁ (B);Read(B)		
B \leftarrow B+100;Write(B);u ₁ (B)	150	
	250	150

Rule #3 Two phase locking (2PL) for transactions

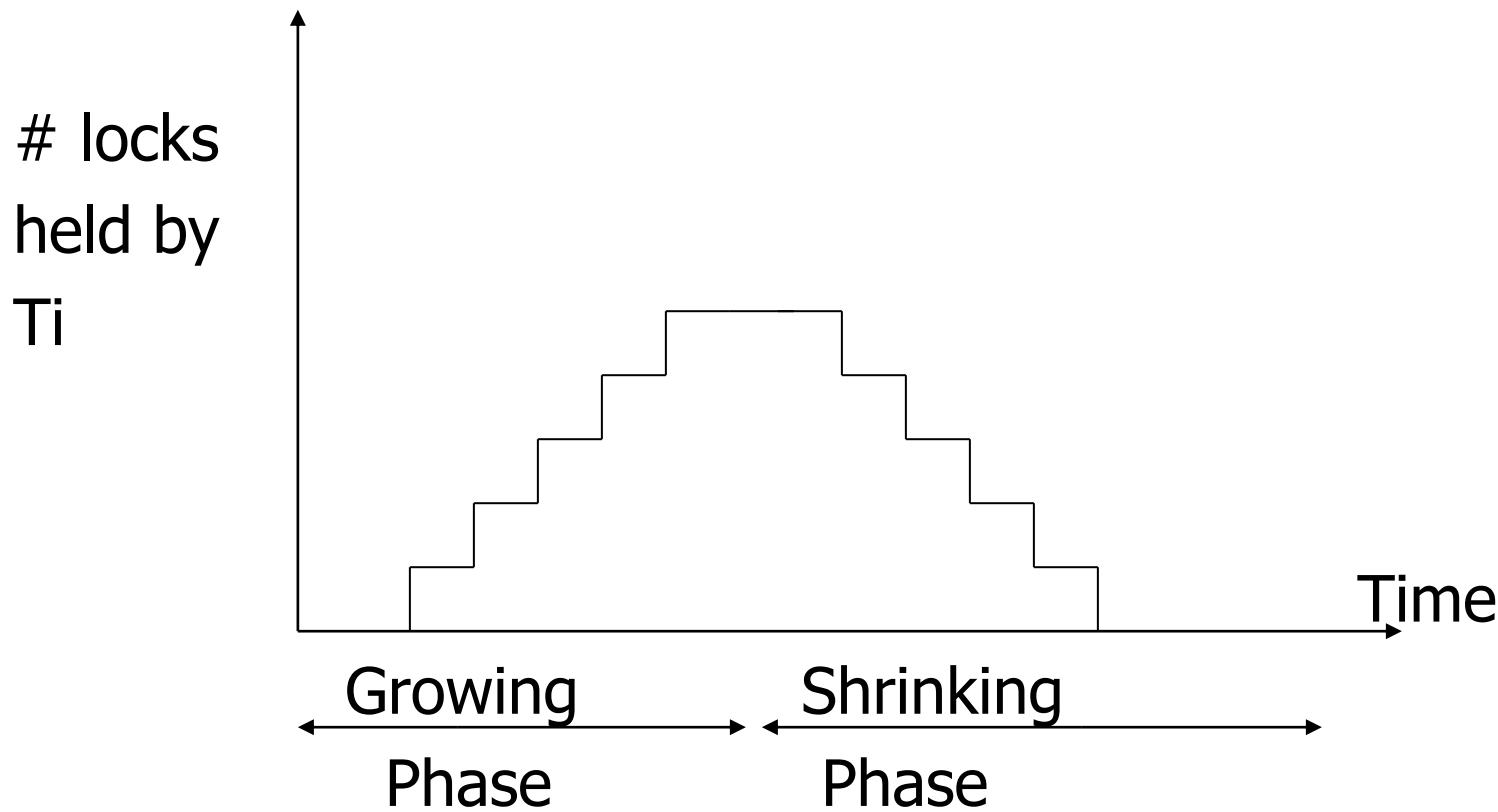
$T_i = \dots \text{ li}(A) \dots \text{ ui}(A) \dots$



no unlocks



no locks



Schedule G

T1

$l_1(A); \text{Read}(A)$

$A \leftarrow A + 100; \text{Write}(A)$

$l_1(B); u_1(A)$

T2

$l_2(A); \text{Read}(A)$

$A \leftarrow Ax2; \text{Write}(A)$

delayed

$l_2(B)$

Schedule G

T1

$l_1(A); \text{Read}(A)$
 $A \leftarrow A + 100; \text{Write}(A)$
 $l_1(B); u_1(A)$

 $\text{Read}(B); B \leftarrow B + 100$
 $\text{Write}(B); u_1(B)$

T2

$l_2(A); \text{Read}(A)$
 $A \leftarrow Ax2; \text{Write}(A);$ *delayed*
 $l_2(B)$

Schedule G

T1

$l_1(A); \text{Read}(A)$
 $A \leftarrow A + 100; \text{Write}(A)$
 $l_1(B); u_1(A)$

$\text{Read}(B); B \leftarrow B + 100$
 $\text{Write}(B); u_1(B)$

T2

$l_2(A); \text{Read}(A)$
 $A \leftarrow Ax2; \text{Write}(A);$ *delayed*
 $l_2(B); u_2(A); \text{Read}(B)$
 $B \leftarrow Bx2; \text{Write}(B); u_2(B);$

Schedule G "good" (equivalent to a serial)

	A	B
T1	25	25
$l_1(A); Read(A); A \leftarrow A + 100$ $Write(A); l_1(B); u_1(A)$	125	
$l_2(A); Read(A)$ $A \leftarrow Ax2; Write(A);$ $l_2(B); delayed$	250	
$Read(B);$ $B \leftarrow B + 100; Write(B); u_1(B)$	125	
$l_2(B); u_2(A); Read(B)$ $B \leftarrow Bx2; Write(B); u_2(B)$	250	250

Theorem Rules #1,2,3 \Rightarrow conflict
(consistency, legality, 2PL) serializable
 schedule

Theorem Rules #1,2,3 \Rightarrow conflict
(consistency, legality, 2PL) serializable
 schedule

Intuitively, each two-phase-locked transaction may be thought to execute in its entirety at the instant it issues its first unlock request. In a conflict-equivalent serial schedule transactions appear **in the same order as their first unlocks**.

Proof:

BASIS: If $n = 1$, there is nothing to do; S is already a serial schedule.

INDUCTION: Suppose S involves n transactions

T_1, T_2, \dots, T_n , and let T_i be the transaction with the first **unlock** action in the entire schedule S , say $u_i(x)$.

We claim it is possible to move all the read and write actions of T_i forward to the beginning of the schedule without passing any conflicting reads or writes.

Theorem Rules #1,2,3 \Rightarrow conflict
(2PL) serializable
schedule

To help in proof:

Lemma

$T_i \rightarrow T_j$ in $S \Rightarrow SH(T_i) <_S SH(T_j)$

Proof of lemma:

$T_i \rightarrow T_j$ means that

$S = \dots p_i(A) \dots q_j(A) \dots; \quad p, q \text{ conflict}$

By rules 1,2:

$S = \dots p_i(A) \dots u_i(A) \dots l_j(A) \dots q_j(A) \dots$



By rule 3: $SH(T_i)$ $SH(T_j)$

So, $SH(T_i) <_S SH(T_j)$

Theorem Rules #1,2,3 \Rightarrow conflict
(2PL) serializable
 schedule

Proof:

(1) Assume $P(S)$ has cycle

$$T_1 \rightarrow T_2 \rightarrow \dots T_n \rightarrow T_1$$

(2) By lemma: $SH(T_1) < SH(T_2) < \dots < SH(T_1)$

(3) Impossible, so $P(S)$ acyclic

(4) $\Rightarrow S$ is conflict serializable

Schedule H (T₂ reversed)

T1	T2
$l_1(A); \text{Read}(A)$	$l_2(B); \text{Read}(B)$
$A \leftarrow A + 100; \text{Write}(A)$	$B \leftarrow B \times 2; \text{Write}(B)$

$l_1(B)$ delayed $l_2(A)$ delayed

Deadlock !!!

- Assume deadlocked transactions are rolled back
 - They have no effect
 - They do not appear in schedule

E.g., Schedule H =



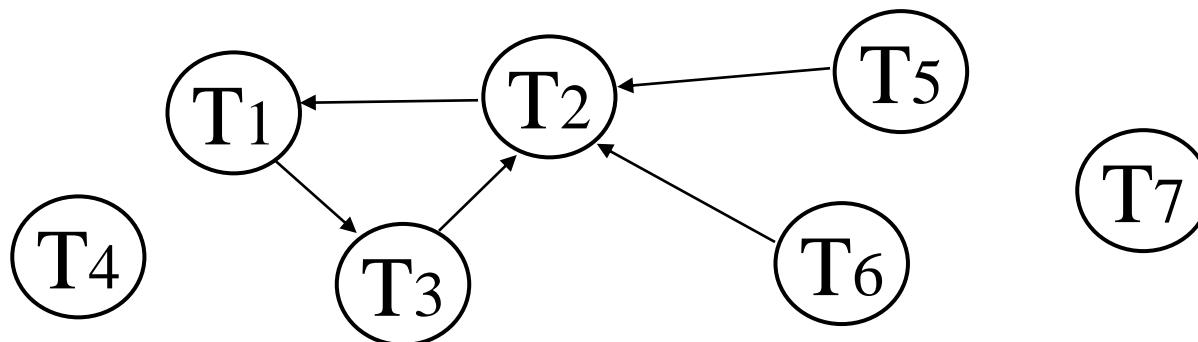
This space intentionally
left blank!

Deadlocks

- **Detection**
 - Wait-for graph
- **Prevention**
 - Resource ordering
 - Timeout
 - Wait-die
 - Wound-wait

Deadlock Detection

- Build **Wait-For graph** ($T_i \rightarrow T_j$ edge if T_i waits for T_j)
- Use lock table structures
- Build incrementally or periodically
- When cycle found, **rollback victim**



Resource Ordering (prevention)

- Order all elements A_1, A_2, \dots, A_n
- A transaction T can lock A_i after A_j only if $i > j$

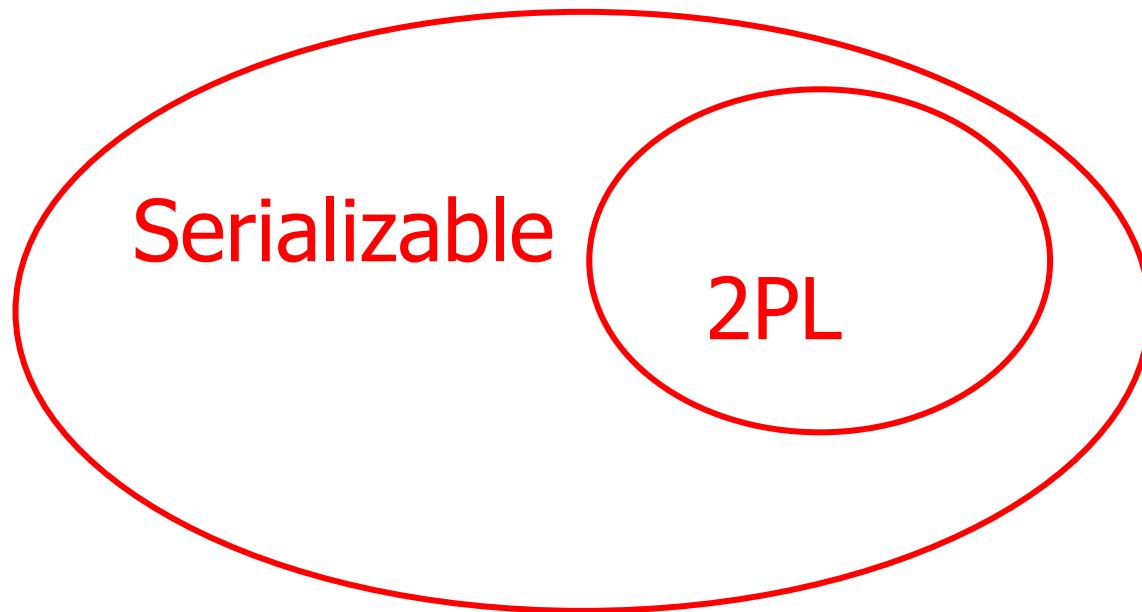
Problem : Ordered lock requests **not realistic** in most cases

Timeout

- If transaction **waits** more than **L sec.**, roll it back!
- Simple scheme
- Hard to select L

2PL subset of Serializable

(schedules that can be implemented by 2PL locks are subset of serializable schedules)



$S1: w1(x) \ w3(x) \ w2(y) \ w1(y)$

- **$S1$ cannot be achieved via 2PL:**
The lock by $T1$ for y must occur after $w2(y)$, so the unlock by $T1$ for x must occur after this point (and before $w3(x)$). Thus, $w3(x)$ cannot occur under 2PL where shown in $S1$ because $T1$ holds the x lock at that point.
- However, **$S1$ is serializable** (conflict-serializable)
(equivalent to $T2, T1, T3$).

If you need a bit more practice:
Are our schedules S_C and S_D 2PL schedules?

S_C : w1(A) w2(A) w1(B) w2(B)

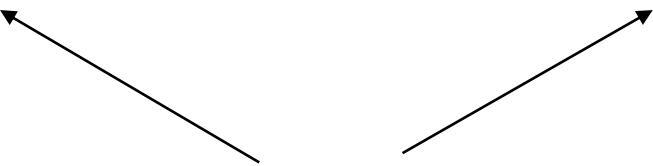
S_D : w1(A) w2(A) w2(B) w1(B)

- Beyond this simple 2PL protocol, it is all a matter of improving performance and **allowing more concurrency**....
 - Shared locks
 - Multiple granularity
 - Inserts, deletes and phantoms
 - Other types of C.C. mechanisms

Shared locks

So far:

$S = \dots l_1(A) r_1(A) u_1(A) \dots l_2(A) r_2(A) u_2(A) \dots$

Do not conflict

Shared locks

So far:

$S = \dots l_1(A) r_1(A) u_1(A) \dots l_2(A) r_2(A) u_2(A) \dots$

Do not conflict

Instead:

$S = \dots ls_1(A) r_1(A) ls_2(A) r_2(A) \dots us_1(A) us_2(A)$

Lock actions

$l-t_i(A)$: lock A in t mode (**t is S or X**)

$u-t_i(A)$: unlock t mode (t is S or X)

Shorthand:

$u_i(A)$: unlock whatever modes

T_i has locked A

Rule #1 Consistency of transactions

$T_i = \dots \mid -S_1(A) \dots r_1(A) \dots u_1(A) \dots$

$T_i = \dots \mid -X_1(A) \dots w_1(A) \dots u_1(A) \dots$

- A transaction **may not write without holding an exclusive lock** and **may not read without holding some lock**.
- All locks must be followed by an unlock of the same element.

Two-phase locking of transactions:

Locking must precede unlocking.

Legality of schedules:

An element may either be locked exclusively by one transaction or by several in shared mode, but not both.

- What about transactions that **read and write same object?**

Option 1: Request **exclusive** lock

$T_i = \dots l-X_1(A) \dots r_1(A) \dots w_1(A) \dots u(A) \dots$

- What about transactions that read and write same object?

Option 2: **Upgrade**

(E.g., need to read, but don't know if it will write...)

$T_i = \dots \text{I-S}_1(A) \dots r_1(A) \dots \text{I-X}_1(A) \dots w_1(A) \dots u(A) \dots$



Think of

- Get 2nd lock on A, or
- Drop S, get X lock

Rule #2 Legal scheduler

$S = \dots I-S_i(A) \dots \dots u_i(A) \dots$



no $I-X_j(A)$

$S = \dots I-X_i(A) \dots \dots u_i(A) \dots$



no $I-X_j(A)$

no $I-S_j(A)$

A way to summarize Rule #2

Compatibility matrix

Comp

	S	X
S	true	false
X	false	false

Rows: held locks

Columns: requested locks

A way to summarize Rule #2

Compatibility matrix

Comp

	S	X
S	true	false
X	false	false

We can grant the lock on X in mode C if and only if for every row R such that there is already a lock on X in mode R by some other transaction, there is a "true" in column C.

Rule # 3 2PL transactions

No change, take care of upgrades:

- (I) If upgrade gets more locks
(e.g., $S \rightarrow \{S, X\}$) then no change!
(It is like a new lock, allowed only in the growing phase)
- (II) If upgrade releases read (shared) lock (e.g., $S \rightarrow X$)
 - can be allowed in growing phase
(We consider it as not a real release or unlock)

Proof: similar to X locks case

Detail:

$| - t_i(A), | - r_j(A)$ (t,r can be a lock mode)

do not conflict if $\text{comp}(t,r) = \text{true}$

Lock types beyond S/X

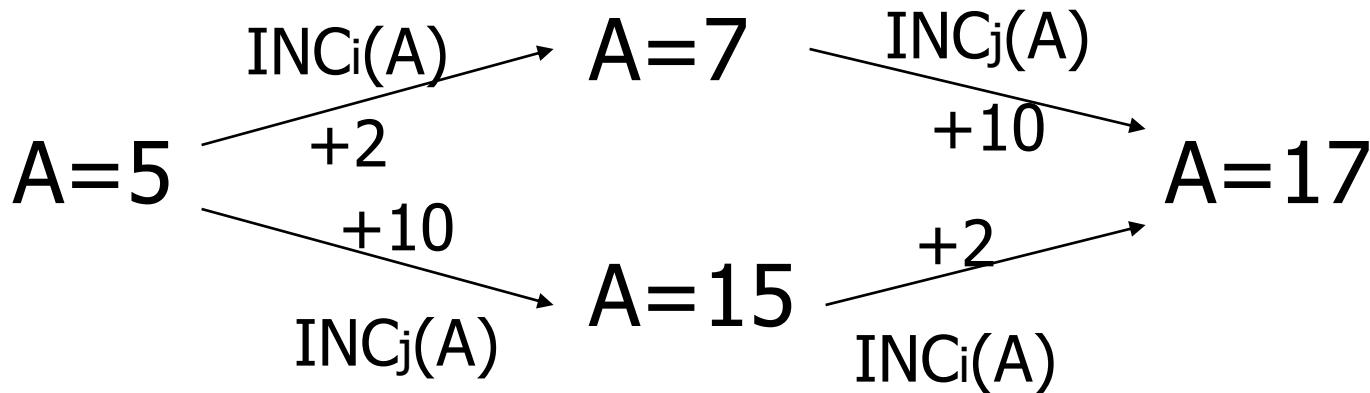
Examples:

- (1) increment lock
- (2) update lock

If we introduce more lock types, it will allow more concurrency (as it happened in case of S/X).

Example (1): increment lock

- Atomic increment action: $\text{INC}_i(A)$
 $\{\text{Read}(A); A \leftarrow A+k; \text{Write}(A)\}$
- $\text{INC}_i(A)$, $\text{INC}_j(A)$ do not conflict!



We need an increment lock

Comp

	S	X	I
S	T	F	F
X	F	F	F
I	F	F	T

Legal schedule: defined by matrix

Any number of transactions can hold an increment lock on X at any time.

If an increment lock on X is held by some transaction, then no other transaction can hold either a shared or exclusive lock on X at the same time.

Upgrade from S -> X

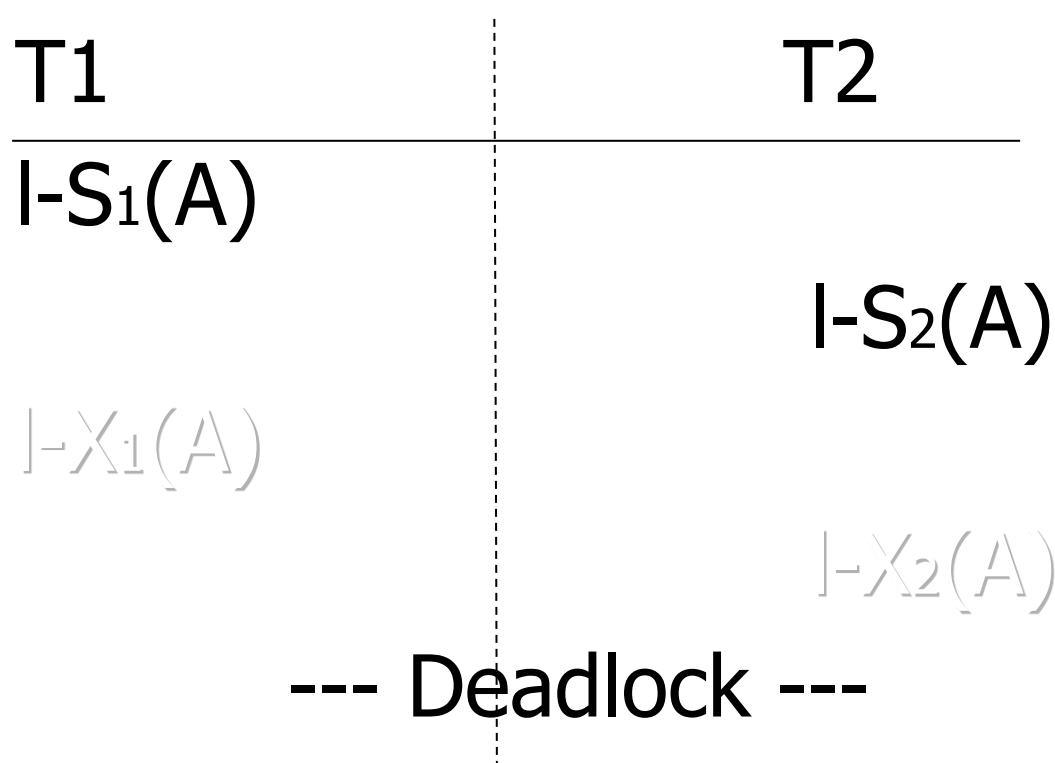
Ti wants to **read A**, does some computation and **write it later**.

Ti locks A in S mode and later **upgrades** the lock to X mode.

It allows more concurrency, other transactions can read A before the upgrade.

Problem with upgrades

A common **deadlock** problem with upgrades:



Solution

If T_i wants to read A and knows it may later want to write A, it requests update lock (not shared).

Rules:

Update lock can be upgraded later to X.
Shared lock cannot be upgraded to X.

New request

Comp

Lock
already
held in

	S	X	U
S	T	F	T
X	F	F	F
U	F	F	F

-> not symmetric table

Note: object A may be **locked in different modes** at the same time...

$$S_1 = \dots \mid - S_1(A) \dots \mid - S_2(A) \dots \mid - U_3(A) \dots \left. \begin{array}{l} \mid - S_4(A) \dots ? \\ \mid - U_4(A) \dots ? \end{array} \right\}$$

- To grant a lock in mode t, mode t **must be compatible with all currently held locks on object** (-> we need a **group mode**)

Group Mode

The group mode is a **summary of the conditions** that a transaction requesting a new lock on A faces.

We can **simplify the grant/deny decision** by comparing the request with only the group mode.

Group mode for S,U -> U

Group mode for S,X -> X

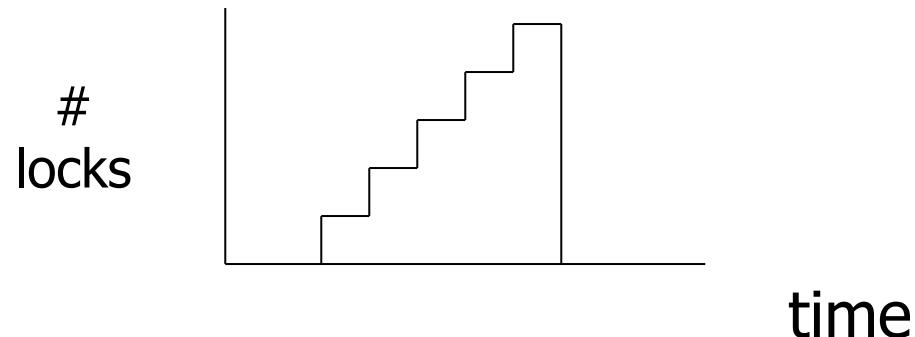
Group mode for U,X -> X

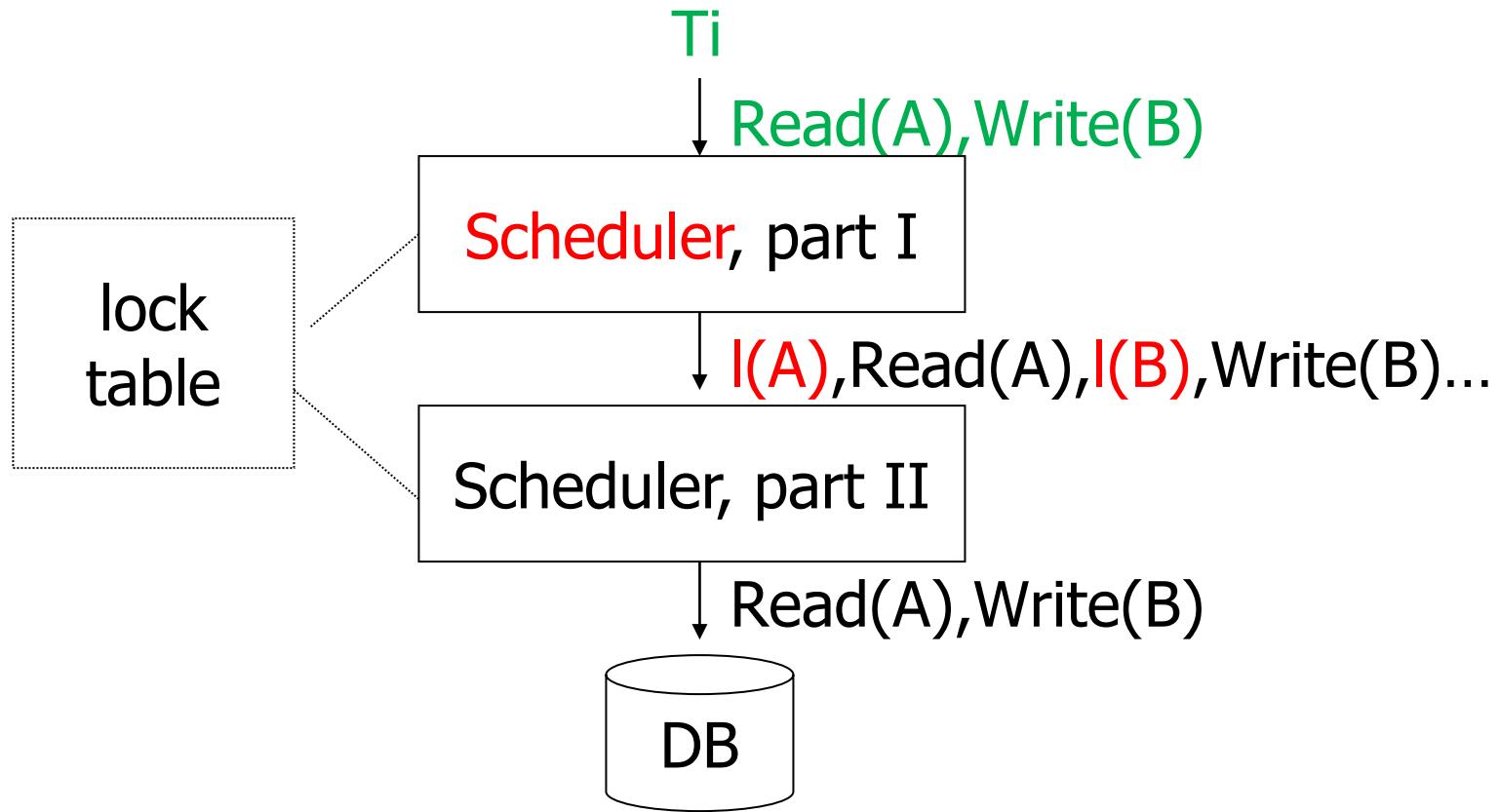
How does locking work in practice?

- Every system is different
(E.g., may not even provide
CONFLICT-SERIALIZABLE schedules)
- But here is one (simplified) way ...

Sample Locking System:

- (1) Don't trust transactions to request/release locks
- (2) Hold all locks until transaction commits

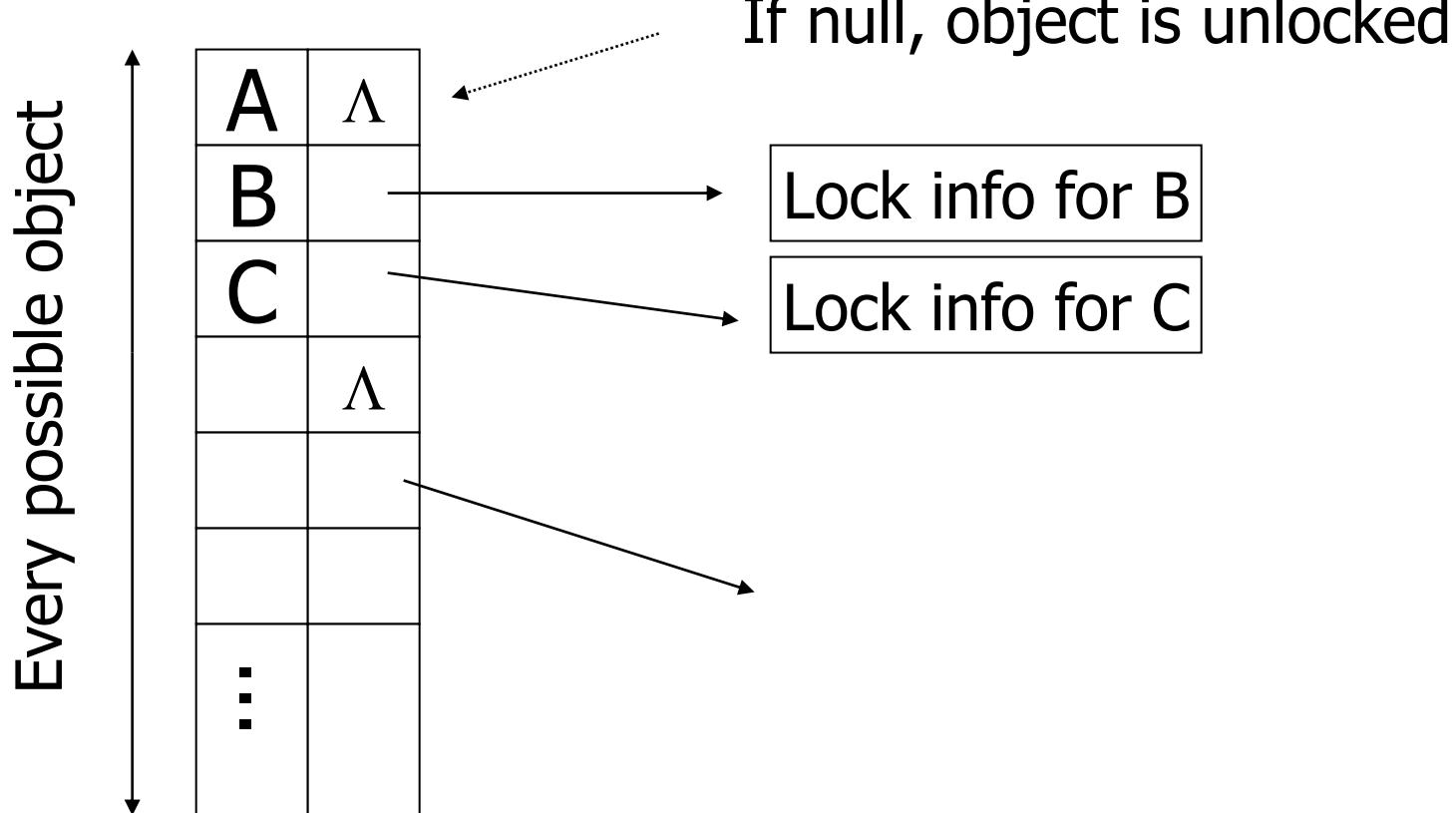




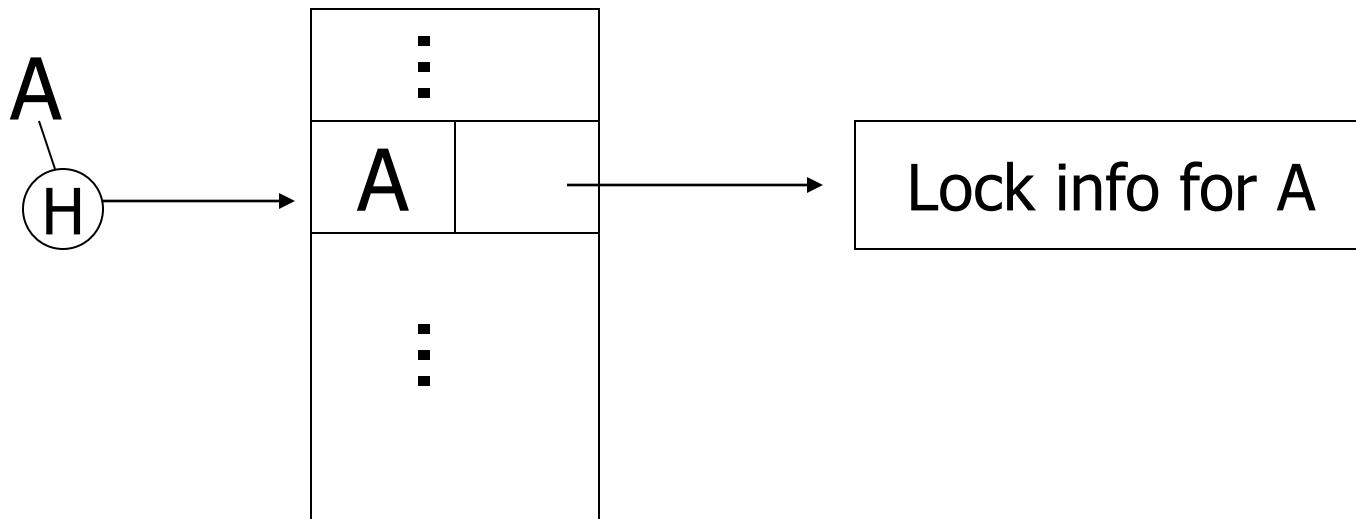
Scheduler requests locks (not transactions)

Locks will be released only at the end of transactions.

Lock table Conceptually

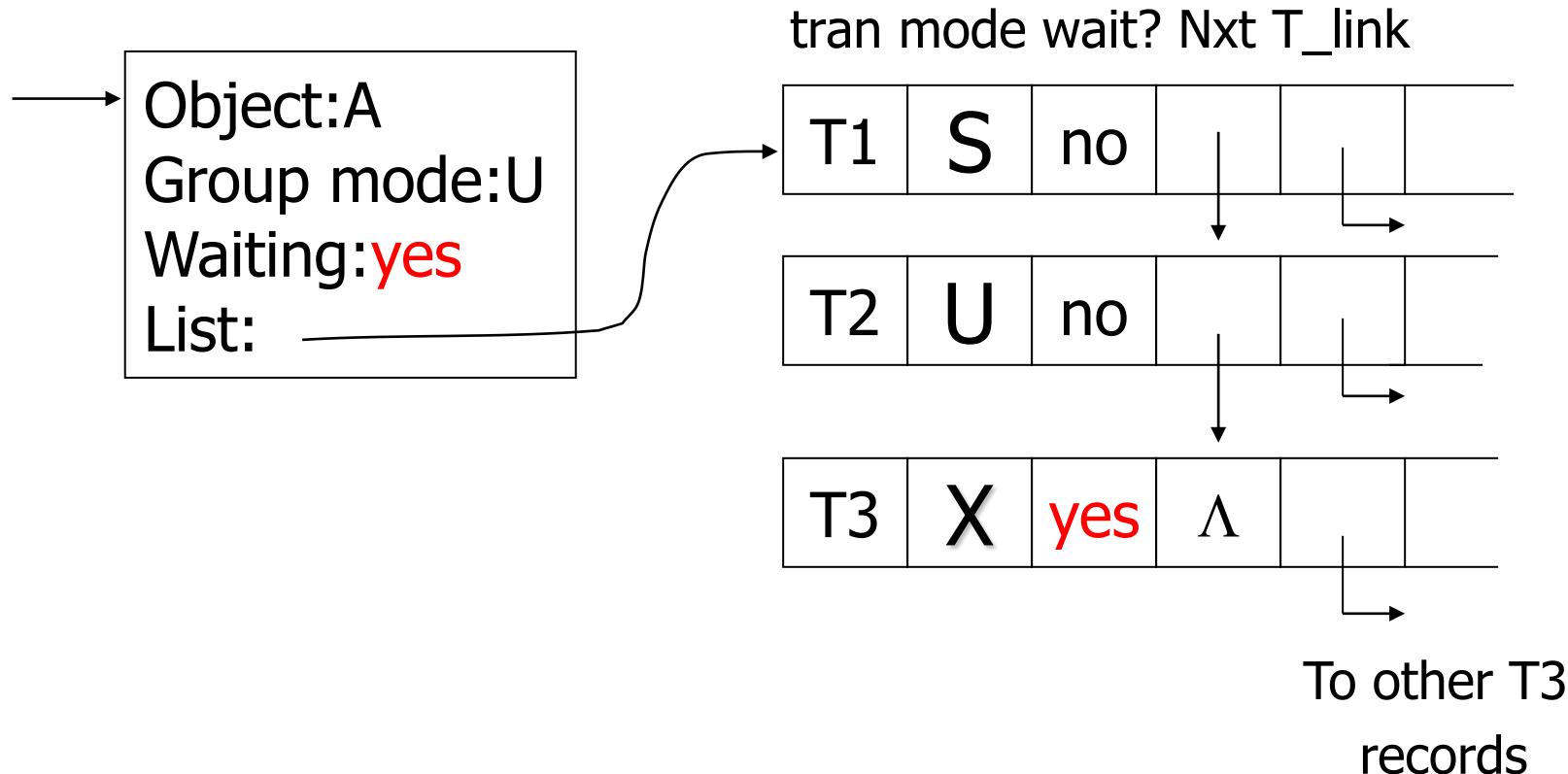


But use hash table:



If object not found in hash table, it is unlocked

Lock info for A - example



Meaning of lock info elements

Group mode:

S – only S locks

U – one U lock, zero or more S locks

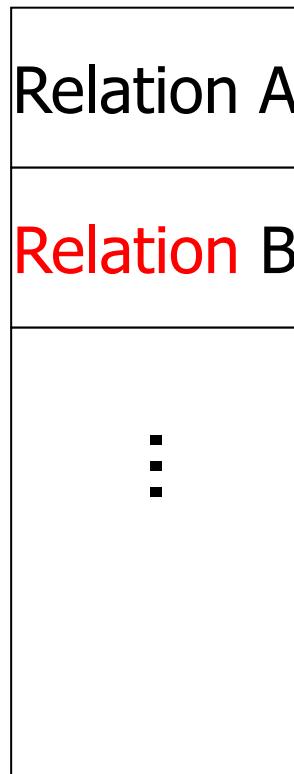
X – one X lock

Waiting: at least 1 transaction waiting for a lock on A

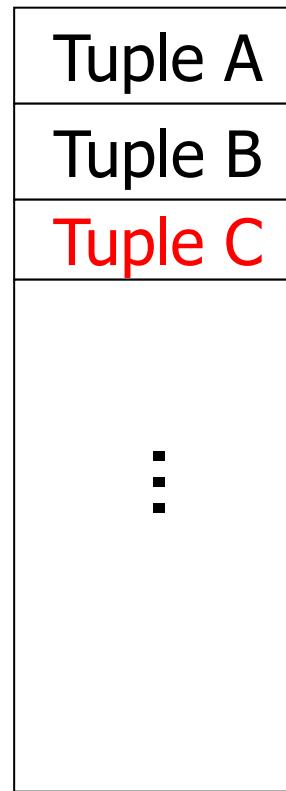
List: transactions waiting for a lock or holding a lock

T_link: to other lock records of the transaction. In case of commit, we can easily release all locks.

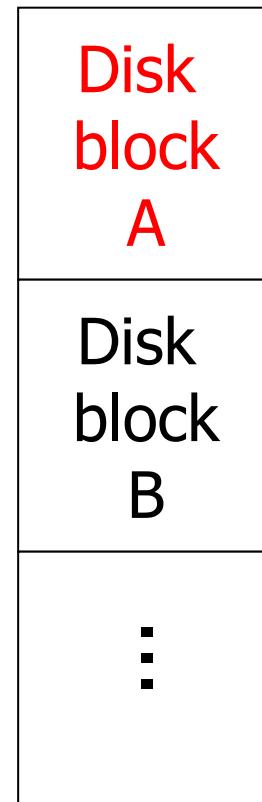
What are the objects we lock?



DB



DB



DB

?

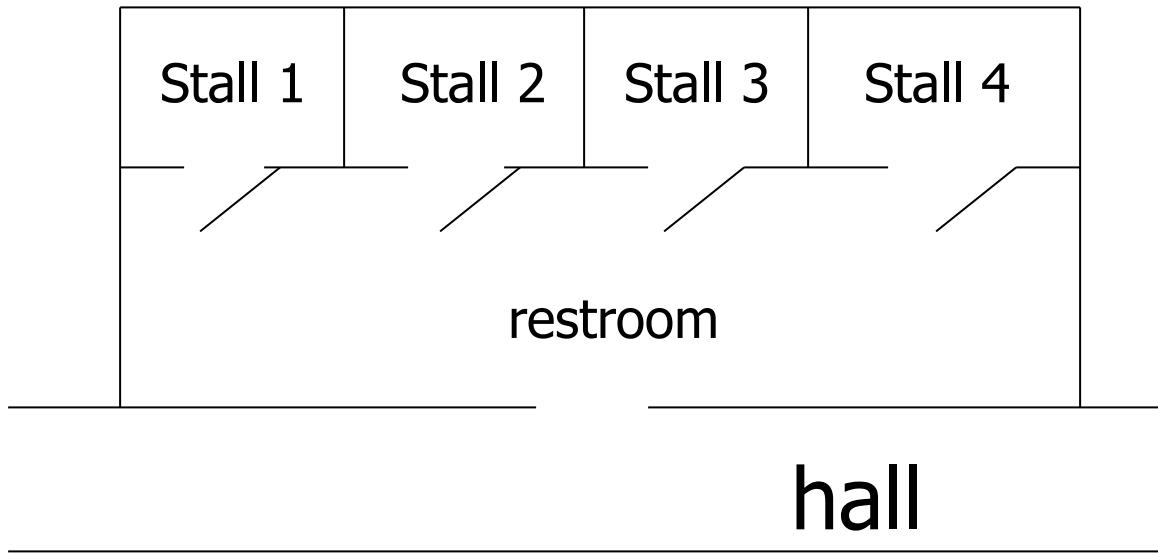
- Locking works in any case, but should we choose small or large objects?

- Locking works in any case, but should we choose small or large objects?
- If we lock **large objects** (e.g., Relations)
 - Need **few locks**
 - **Low concurrency**
- If we lock **small objects** (e.g., tuples, fields)
 - Need **more locks**
 - **More concurrency**

We can have it **both ways!!**

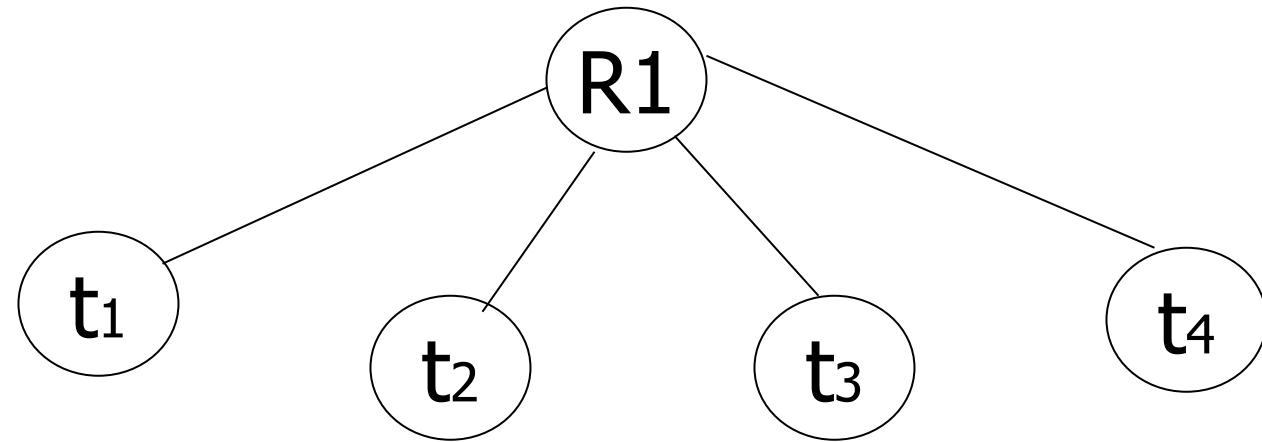
Ask any janitor to give you the solution...

Should we close (**lock**) individual doors or restroom?

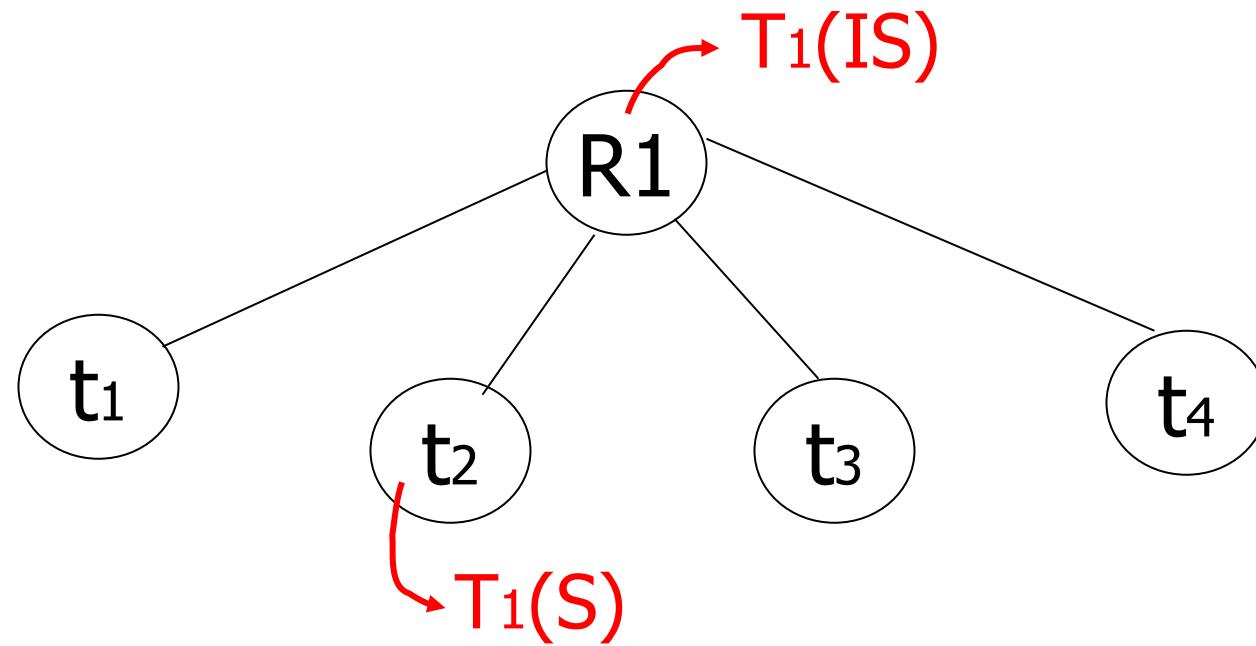


Try to reduce the number of conflicts !!!

Example (R: relation, t: tuple)

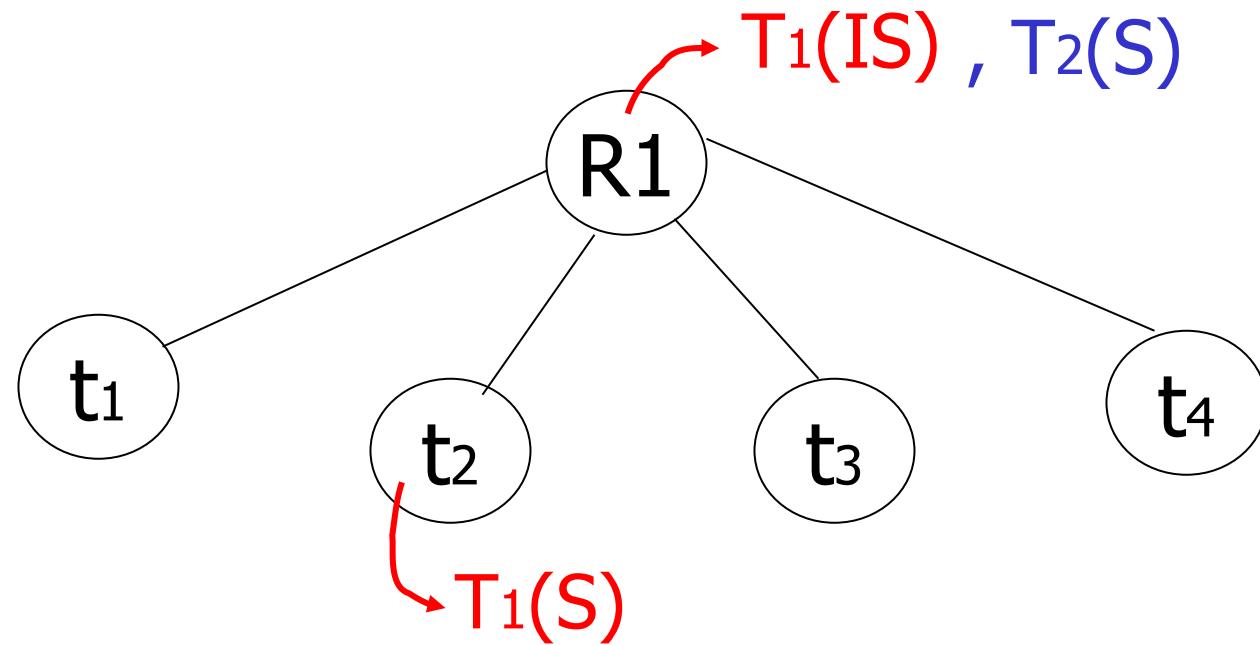


Example (Warning or Intention)

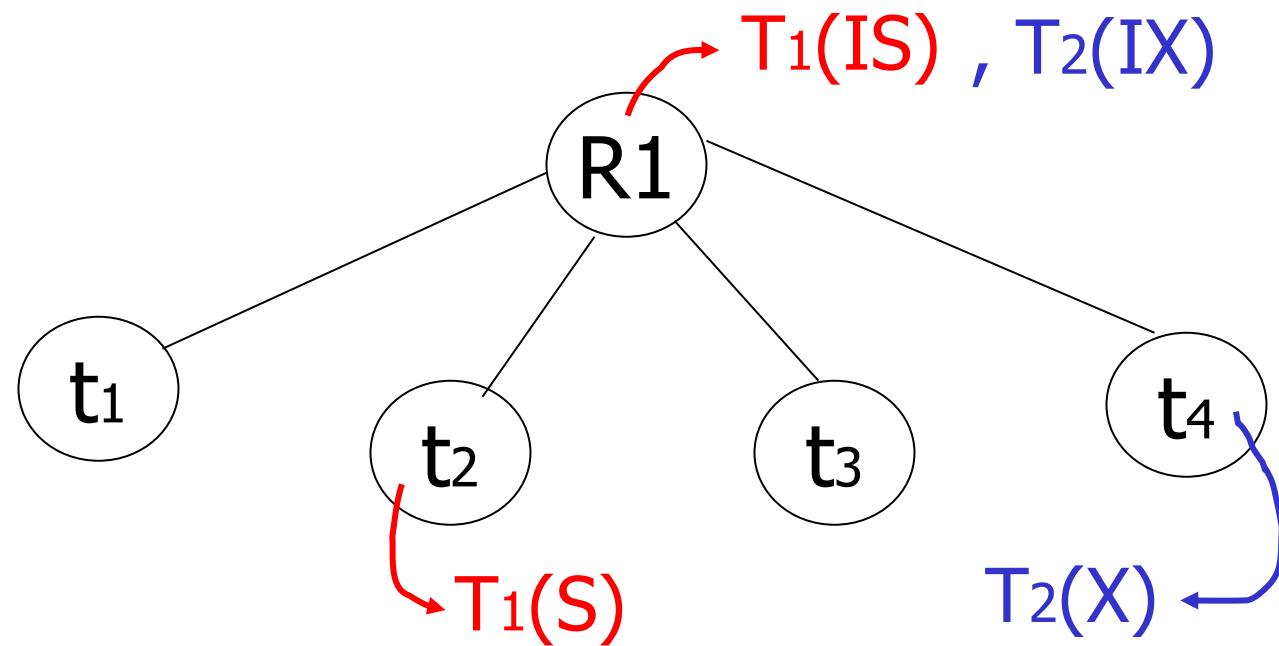


IS: Intention to obtain a shared lock on a subelement
IX: Intention to obtain exclusive lock on a subelement

Example



Example (b)



Multiple granularity (SIX: group mode)

Comp	Requestor				
	IS	IX	S	SIX	X
Holder	IS				
	IX				
	S				
	SIX				
	X				

SIX: group mode for S and IX

Multiple granularity (SIX: group mode)

Comp

Requestor

IS IX S SIX X

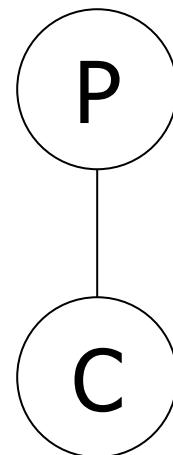
Holder

	IS	IX	S	SIX	X
IS	T	T	T	T	F
IX	T	T	F	F	F
S	T	F	T	F	F
SIX	T	F	F	F	F
X	F	F	F	F	F

Parent
locked in

Child can be locked
by same transaction in

IS	IS, S
IX	IS, S, IX, X, SIX
S	none
SIX	X, IX, [SIX]
X	none



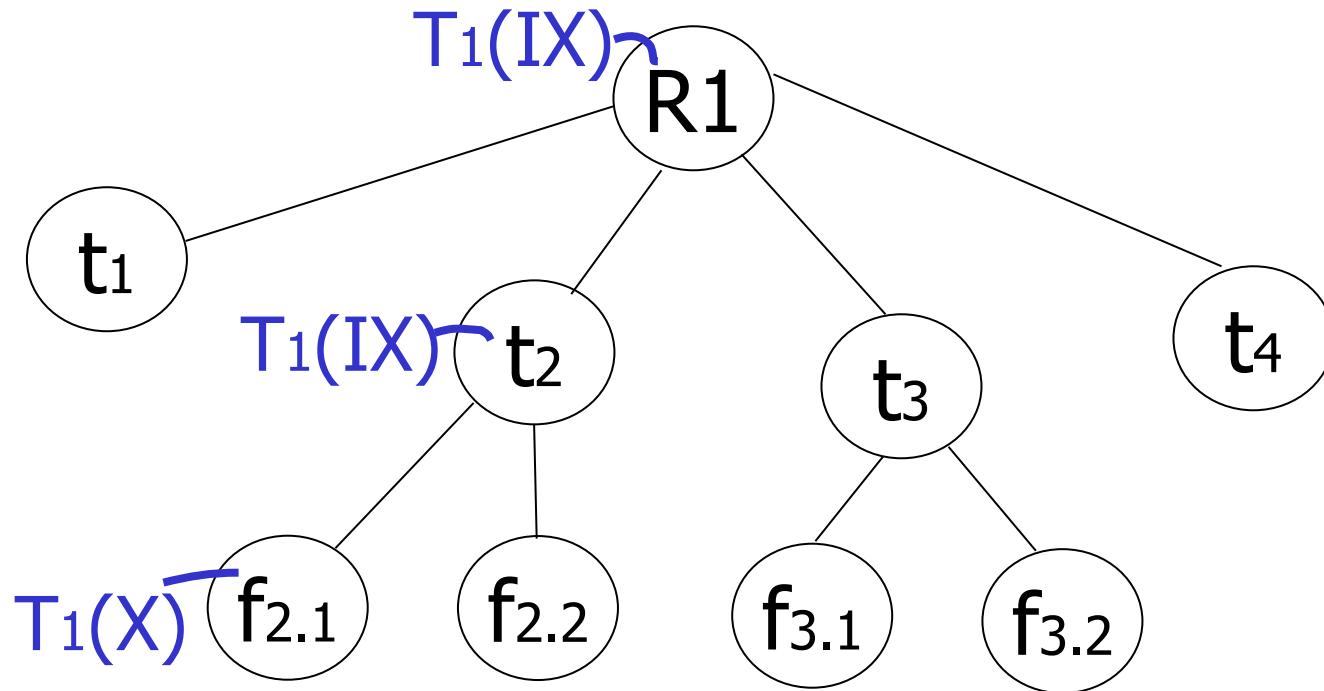
not necessary

Rules

- (1) Follow multiple granularity compat.
- (2) **Lock root** of tree **first**, any mode
- (3) Node Q can be locked by Ti in S or IS only if parent(Q) locked by Ti in IX or IS
- (4) Node Q can be locked by Ti in X,SIX,IX only if parent(Q) locked by Ti in IX,SIX
- (5) Ti is two-phase
- (6) Ti can **unlock** node Q **only if none of Q's children are locked** by Ti

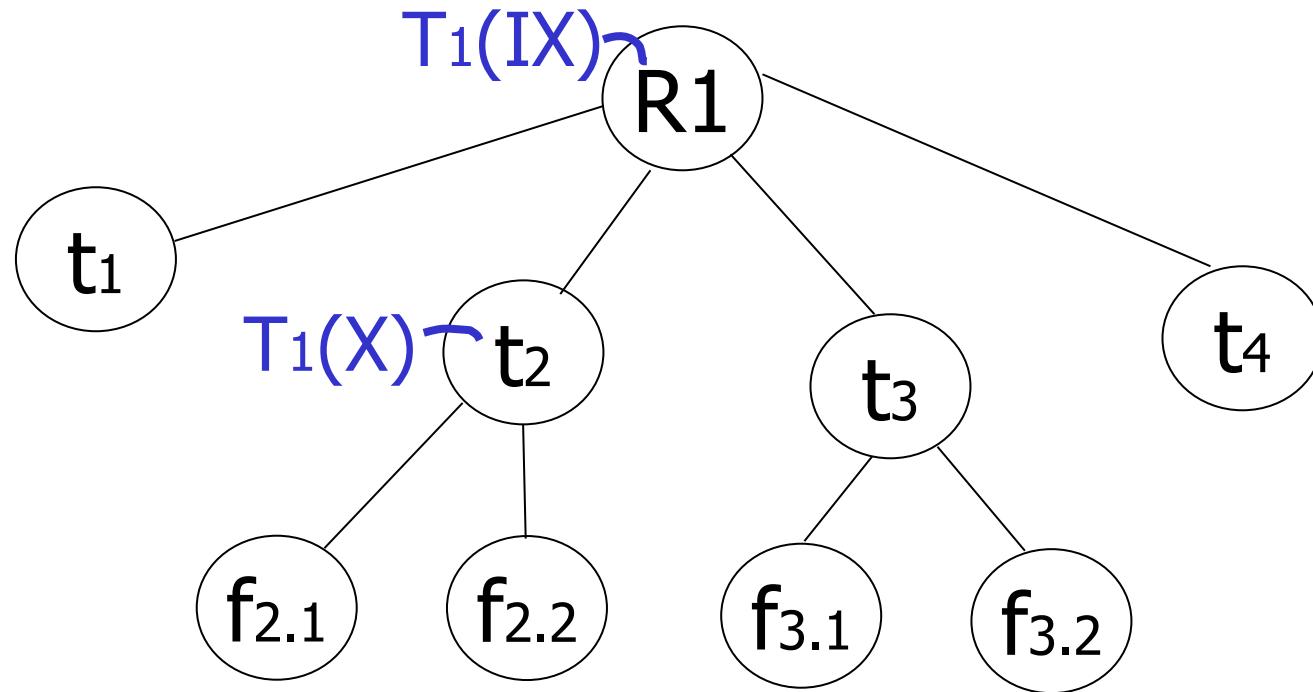
Exercise:

- Can T₂ access object f_{2.2} in X mode?
What locks will T₂ get?



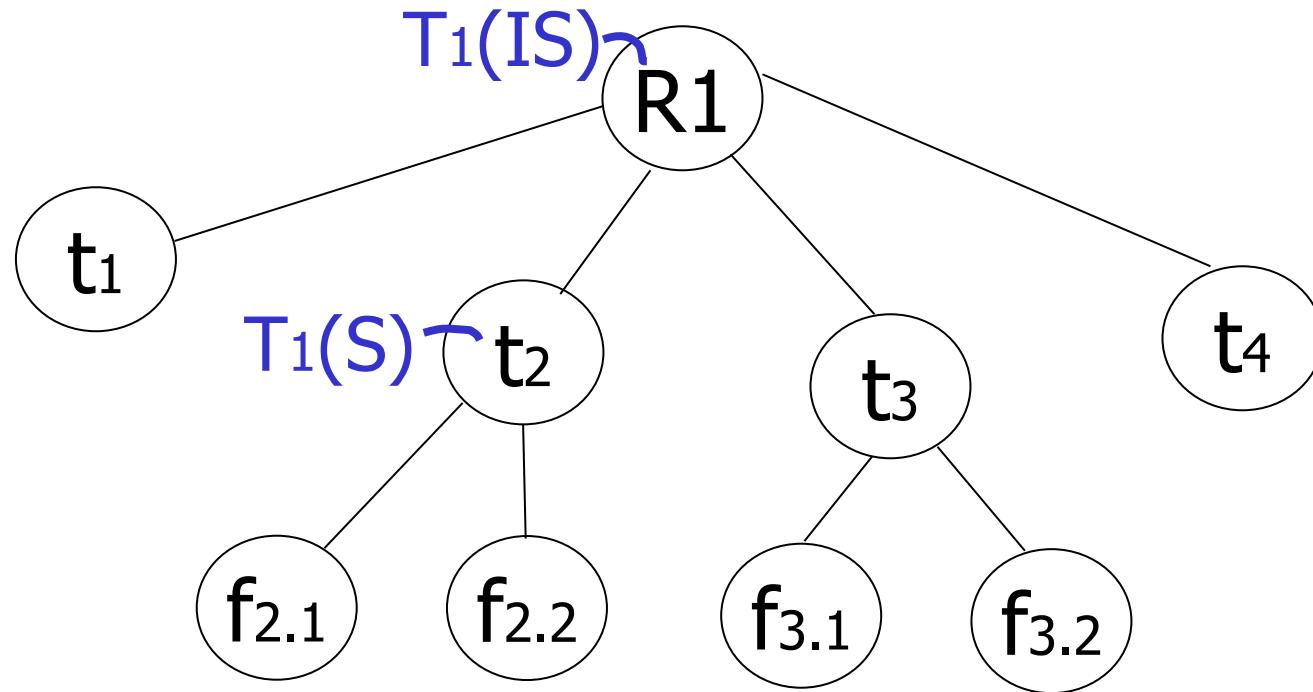
Exercise:

- Can T₂ access object f_{2.2} in X mode?
What locks will T₂ get?



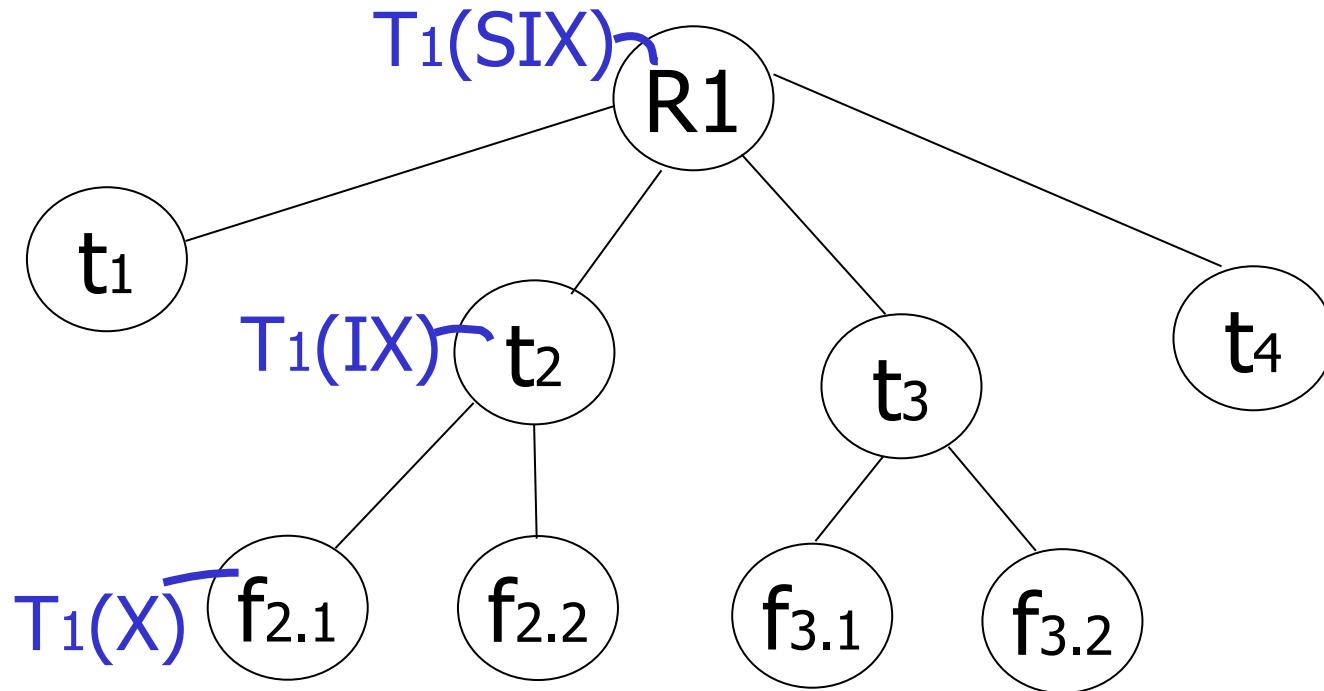
Exercise:

- Can T₂ access object f_{3.1} in X mode?
What locks will T₂ get?



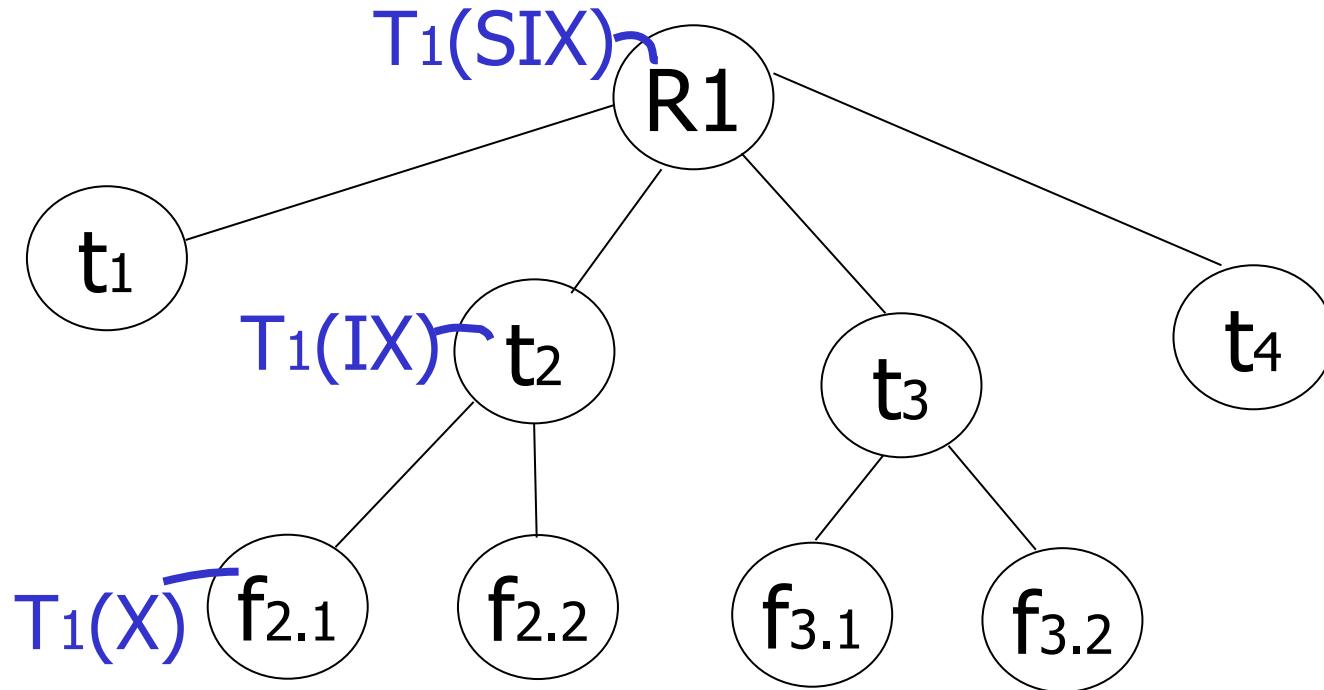
Exercise:

- Can T₂ access object f_{2.2} in S mode?
What locks will T₂ get?

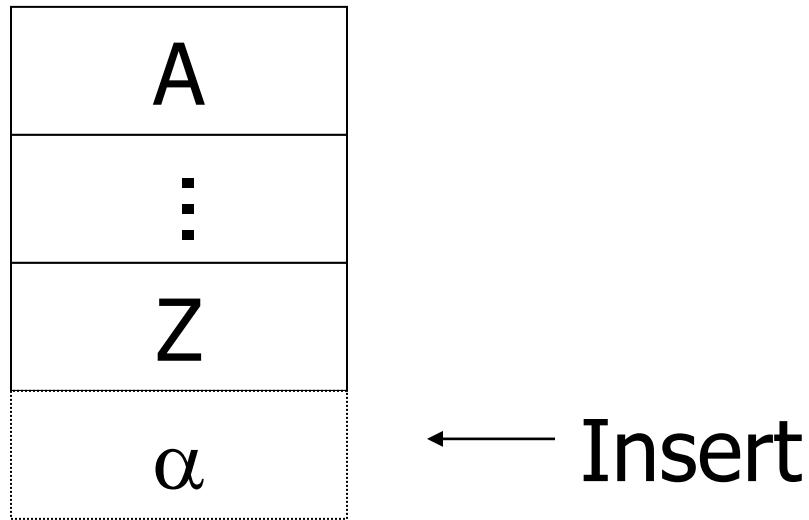


Exercise:

- Can T₂ access object f_{2.2} in X mode?
What locks will T₂ get?



Insert + delete operations



Problem: we can lock only existing elements

Still have a problem: **Phantoms**

Example: relation R (E#,name,...)

constraint: E# is key

use tuple locking

R	E#	Name
r1	55	Smith	
r2	75	Jones	

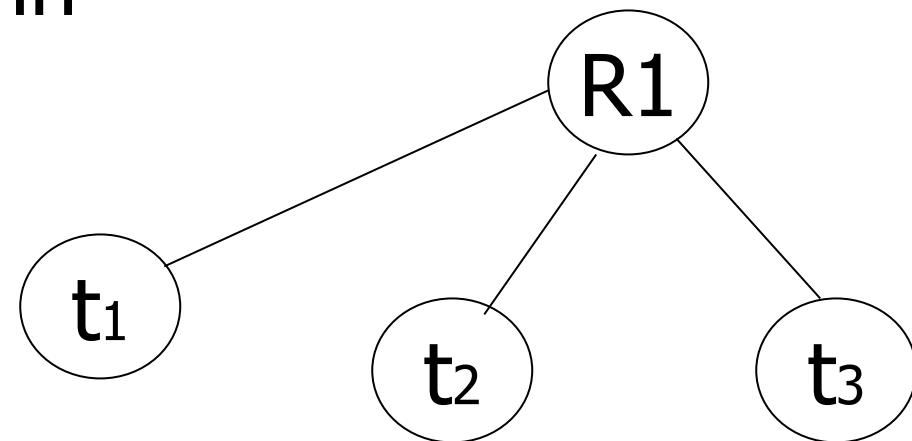
T₁: Insert <08,Obama,...> into R

T₂: Insert <08,McCain,...> into R

T ₁	T ₂
S ₁ (r ₁) (reads a tuple)	S ₂ (r ₁)
S ₁ (r ₂)	S ₂ (r ₂)
Check Constraint	Check Constraint
:	:
Insert r ₃ [08,Obama,...]	Insert r ₄ [08,McCain,...]

Solution

- Use multiple granularity tree
- Before insert of node Q,
lock parent(Q) in
X mode



Modifications to locking rules:

- (1) Get **exclusive lock** on A before deleting from A
- (2) At insert into A operation by T_i , T_i is given **exclusive lock** on A

Back to example

T1: Insert<08,Obama>

T1

$X_1(R)$ (exclusive)

Check constraint

Insert<08,Obama>

U(R) (unlock)

T2: Insert<08,McCain>

T2

$X_2(R)$

delayed

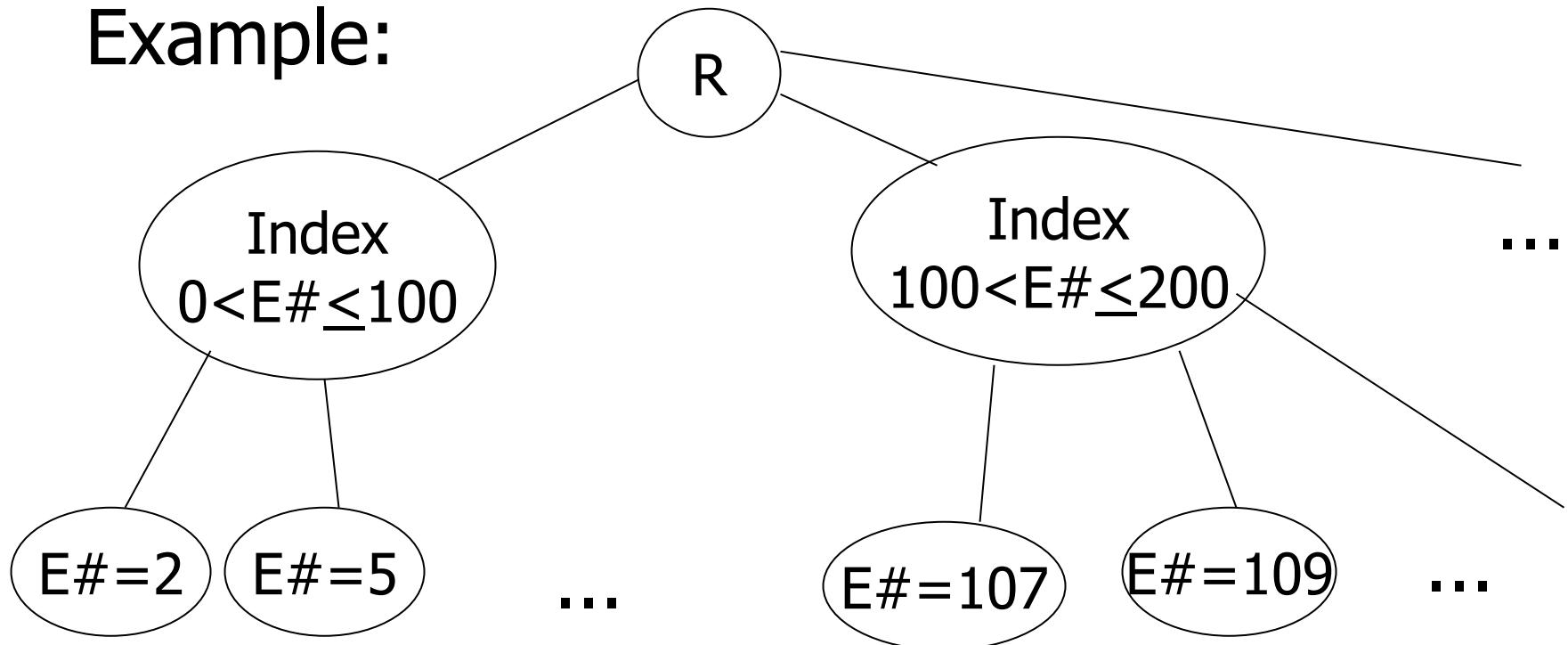
$X_2(R)$

Check constraint

Oops! $e\# = 08$ already in R!

Instead of using R, can use index on R:

Example:



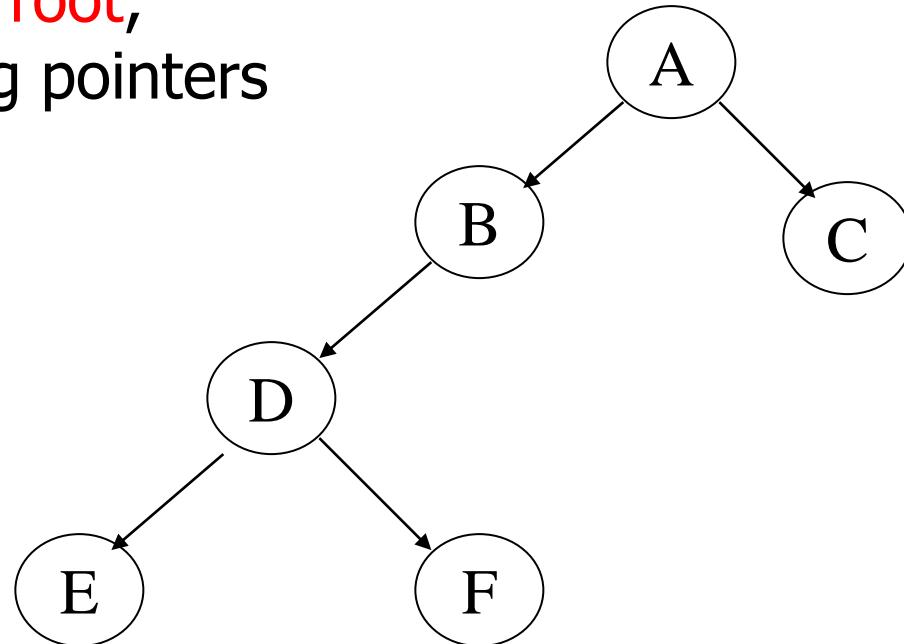
- This approach can be generalized to multiple indexes...

Next:

- Tree-based concurrency control
- Validation concurrency control

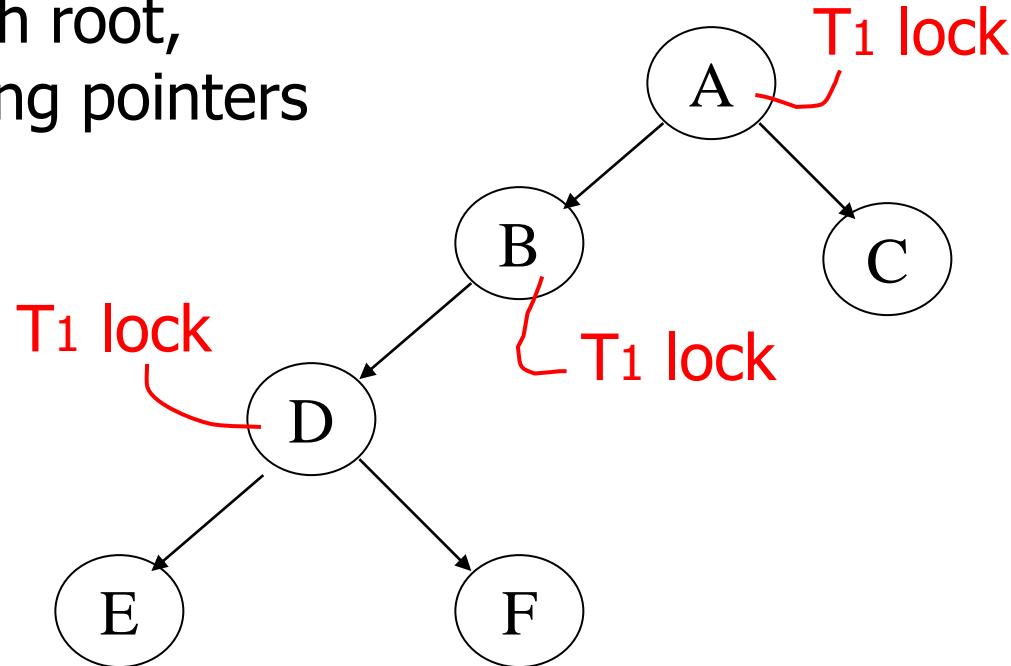
Example

- all objects accessed through root, following pointers



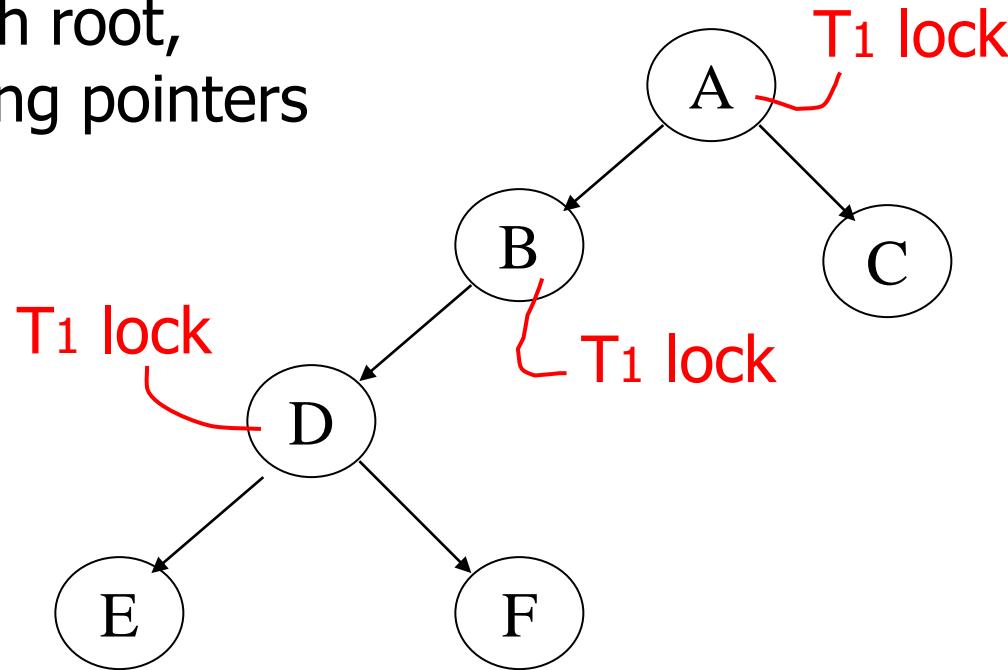
Example

- all objects accessed through root, following pointers



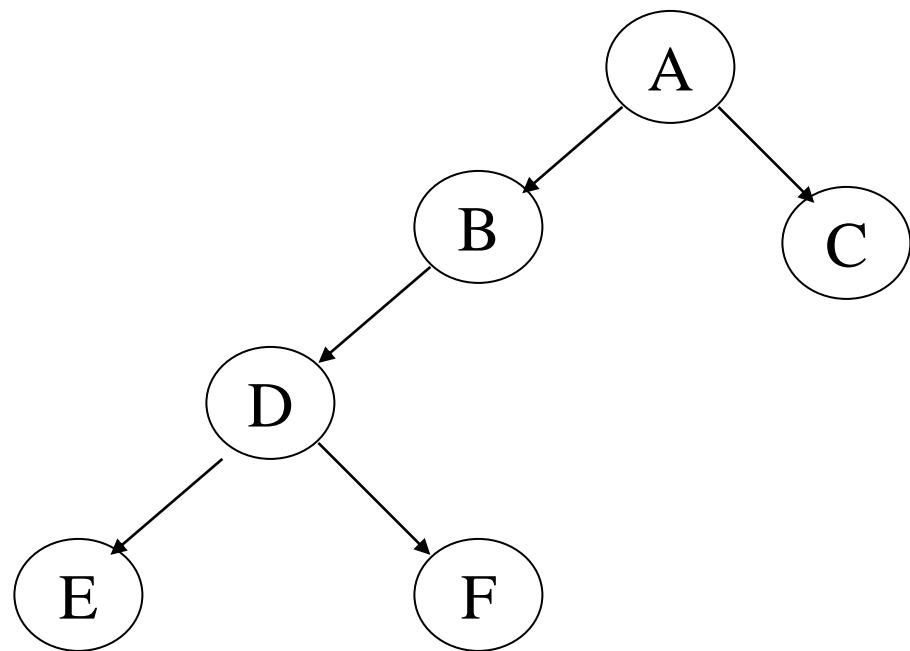
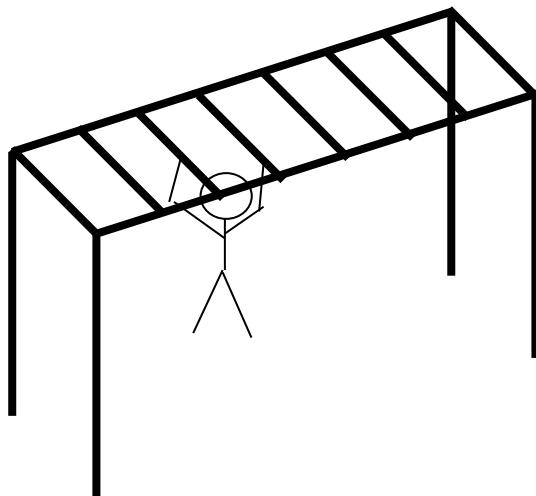
Example

- all objects accessed through root, following pointers

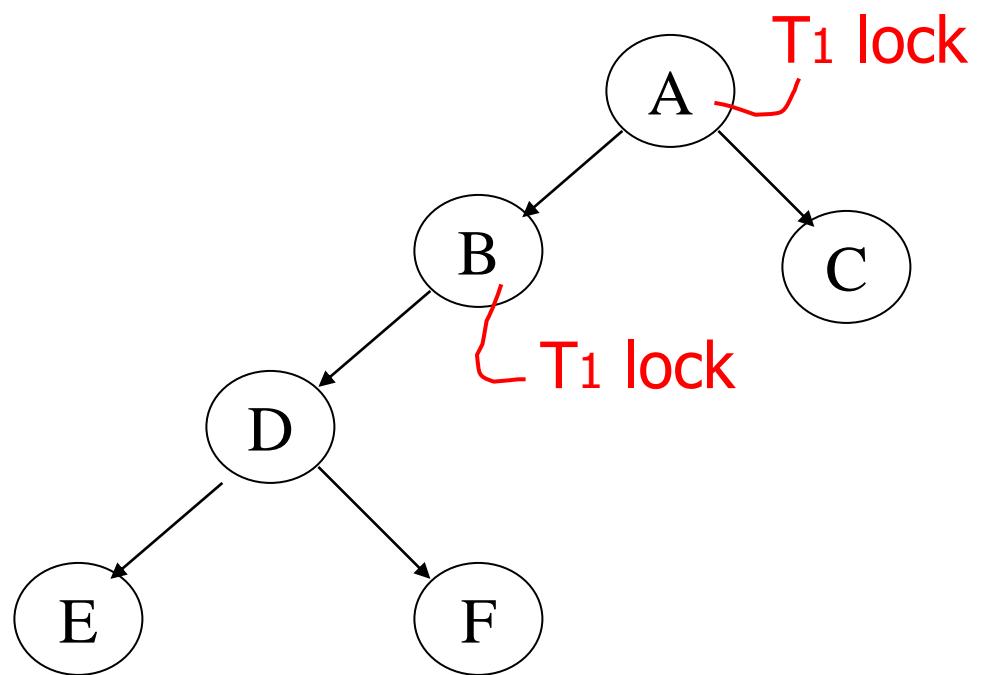
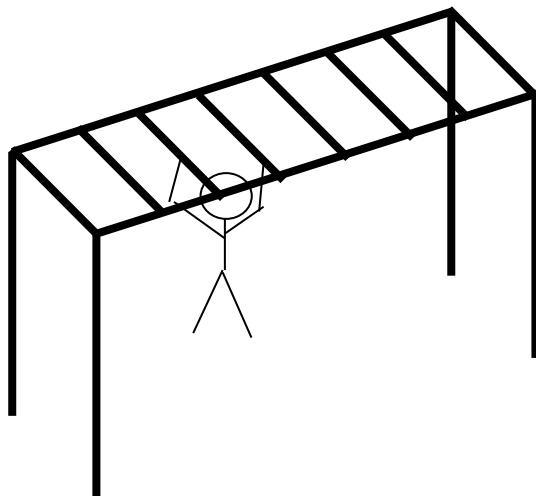


- can we release A lock if we no longer need A?

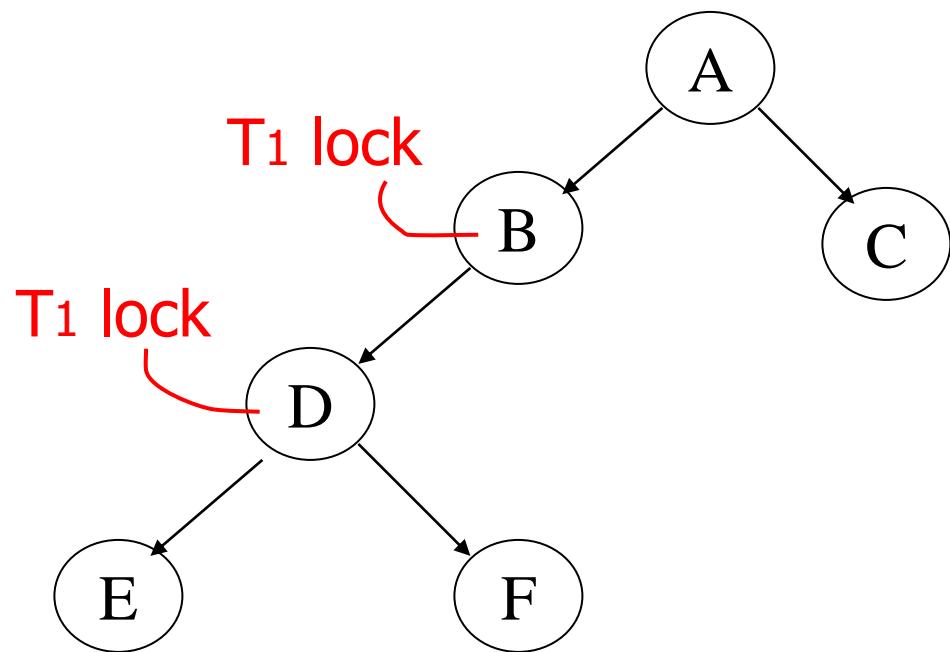
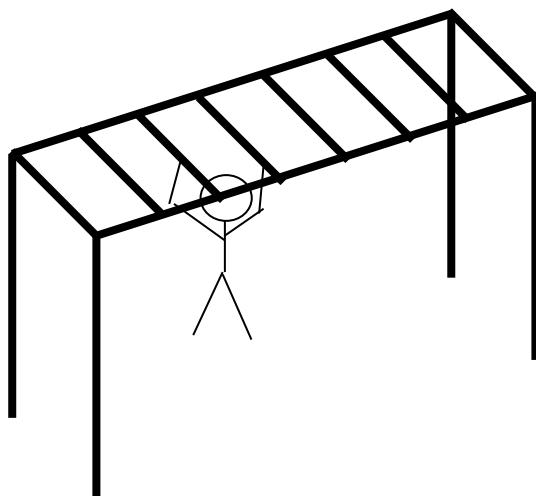
Idea: traverse like “Monkey Bars”



Idea: traverse like “Monkey Bars”

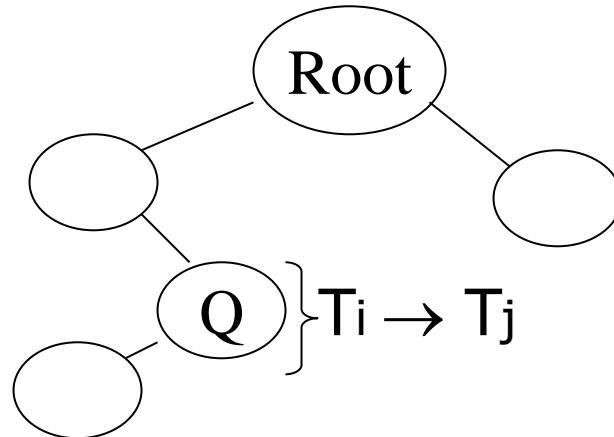


Idea: traverse like “Monkey Bars”



Why does this work?

- Assume all T_i start at root; exclusive lock
- $T_i \rightarrow T_j \Rightarrow T_i$ locks root before T_j

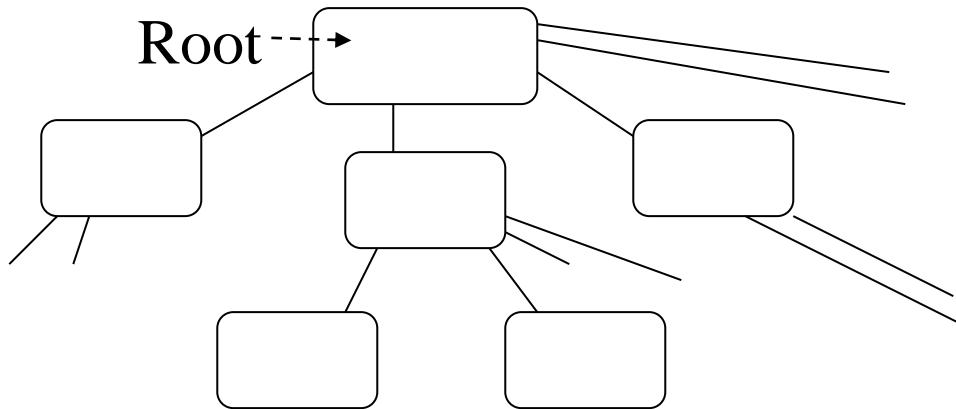


- Actually works if we don't always start at root

Rules: tree protocol (exclusive locks)

- (1) First lock by T_i may be on any item
- (2) After that, item Q can be locked by T_i only if $\text{parent}(Q)$ locked by T_i
- (3) Items may be unlocked at any time
- (4) After T_i unlocks Q , it cannot relock Q
Even if it holds a lock on $\text{parent}(Q)$

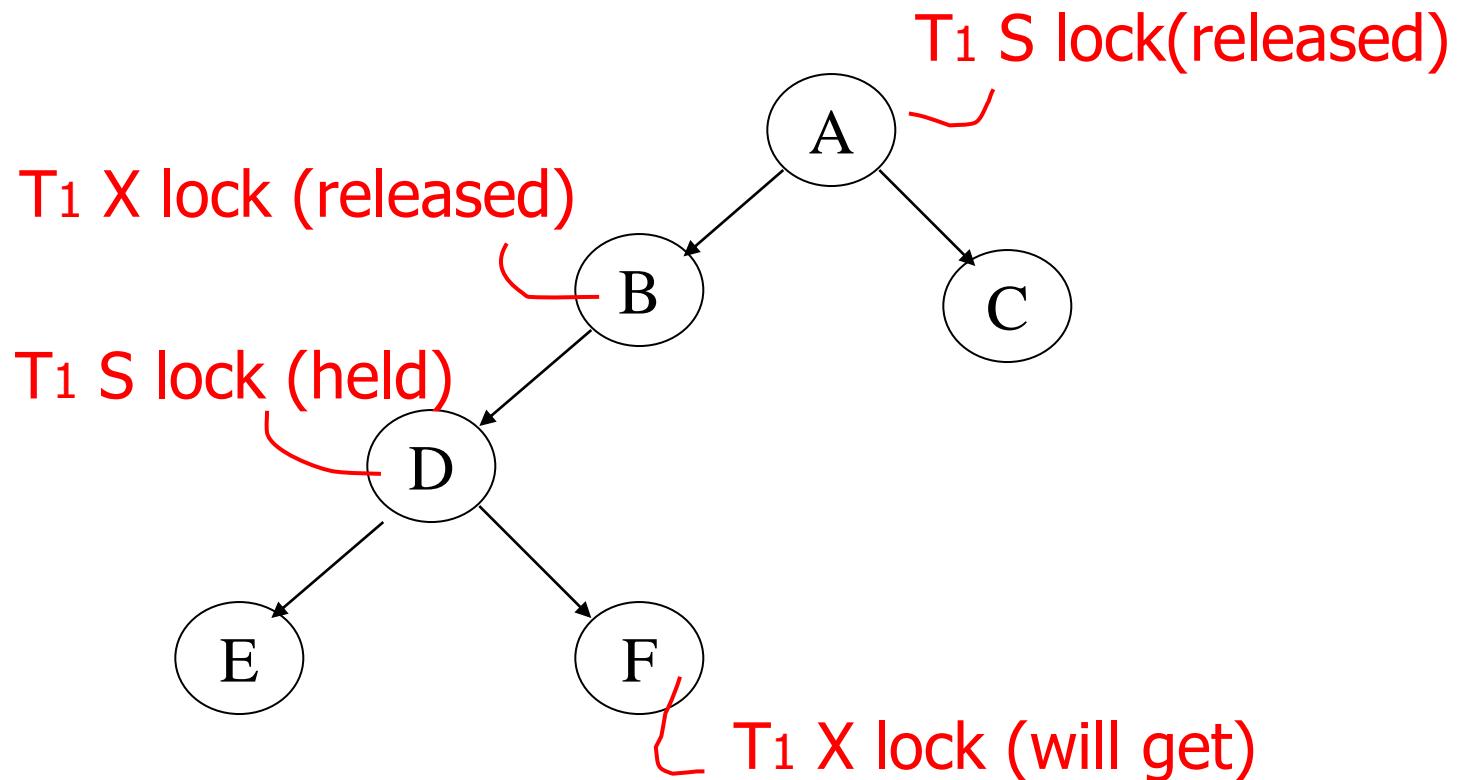
- Tree-like protocols are **used typically for B-tree** concurrency control



E.g., during insert, **do not release parent lock, until you are certain child does not have to split**

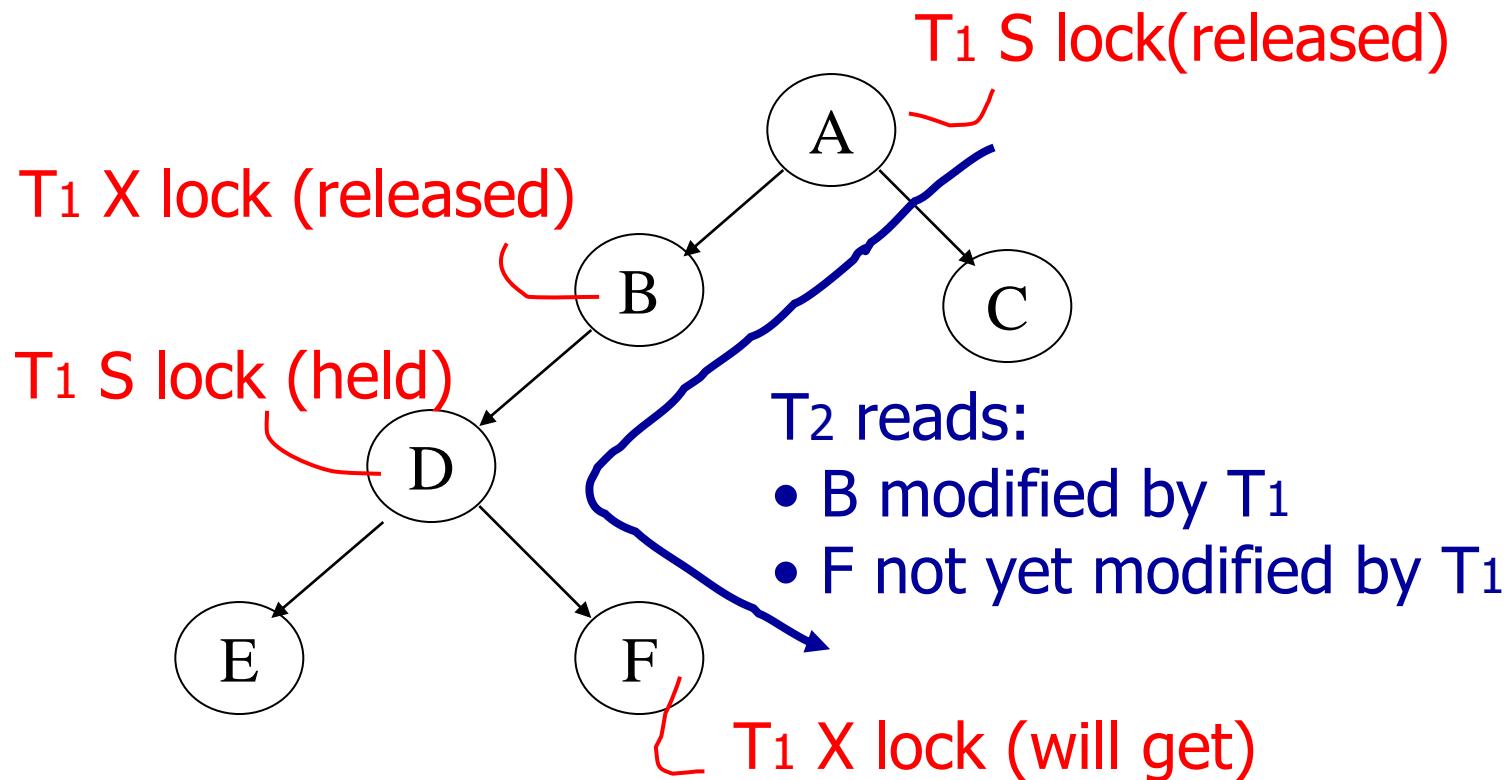
Tree Protocol with Shared Locks

- Rules for shared & exclusive locks?



Tree Protocol with Shared Locks

- Rules for shared & exclusive locks?



Tree Protocol with Shared Locks

- Need more restrictive protocol
- Will this work?
 - Once T_1 locks one object in X mode, all further locks down the tree must be in X mode

Validation

Transactions have 3 phases:

(1) Read

- all DB values read
- writes to temporary storage
- no locking

(2) Validate

- check if schedule so far is serializable

(3) Write

- if validate ok, write to DB

Key idea

- Make validation atomic
- If T_1, T_2, T_3, \dots is validation order, then resulting schedule will be conflict equivalent to $S_s = T_1 T_2 T_3 \dots$

To implement validation, system keeps two sets:

- FIN = transactions that have finished phase 3 (and are all done)
- VAL = transactions that have successfully finished phase 2 (validation)

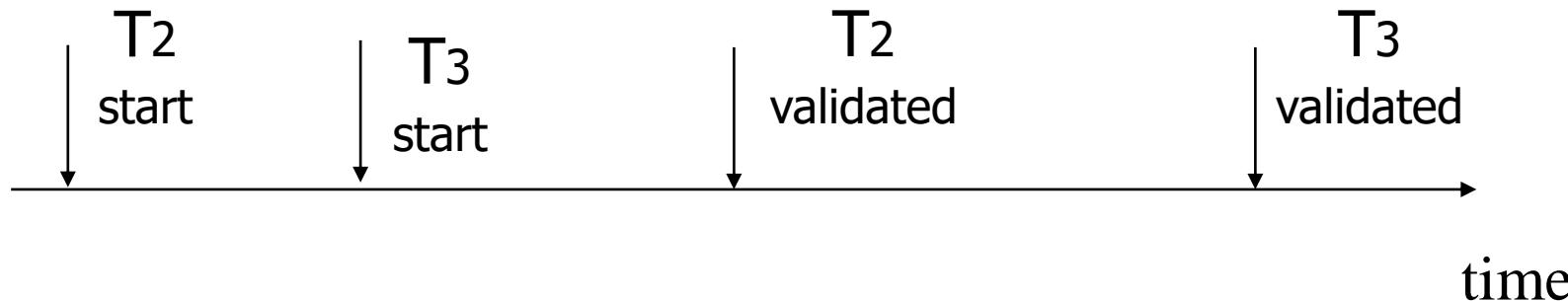
Example of what validation must prevent:

$$RS(T_2) = \{B\}$$

$$WS(T_2) = \{B, D\}$$

$$RS(T_3) = \{A, B\} \neq \emptyset$$

$$WS(T_3) = \{C\}$$



Example of what validation must prevent:

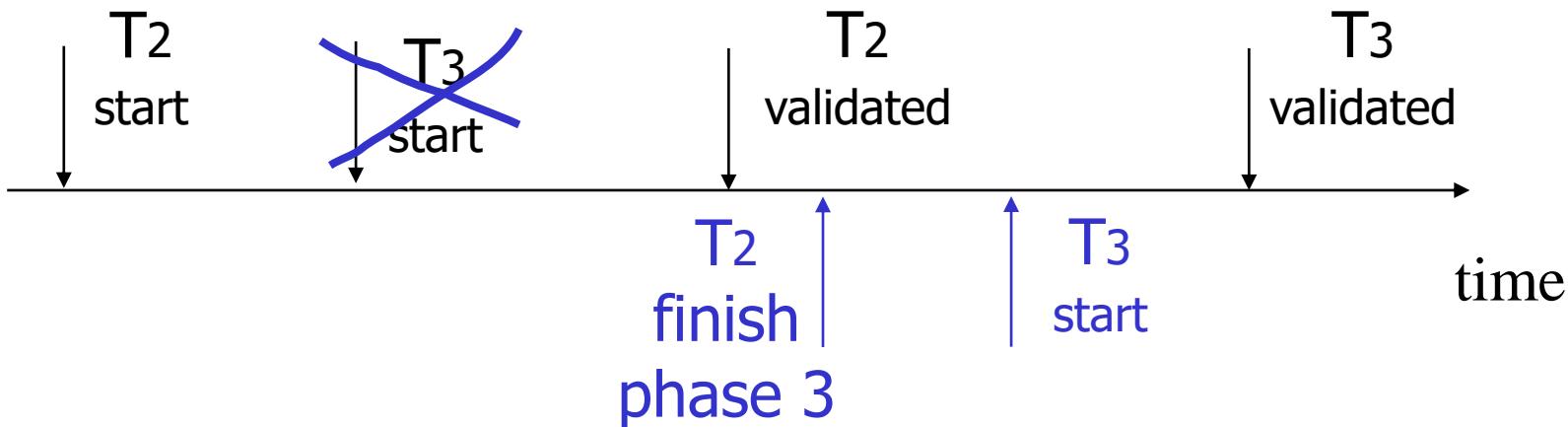
allow

$$RS(T_2) = \{B\}$$

$$WS(T_2) = \{B, D\}$$

$$RS(T_3) = \{A, B\} \neq \emptyset$$

$$WS(T_3) = \{C\}$$



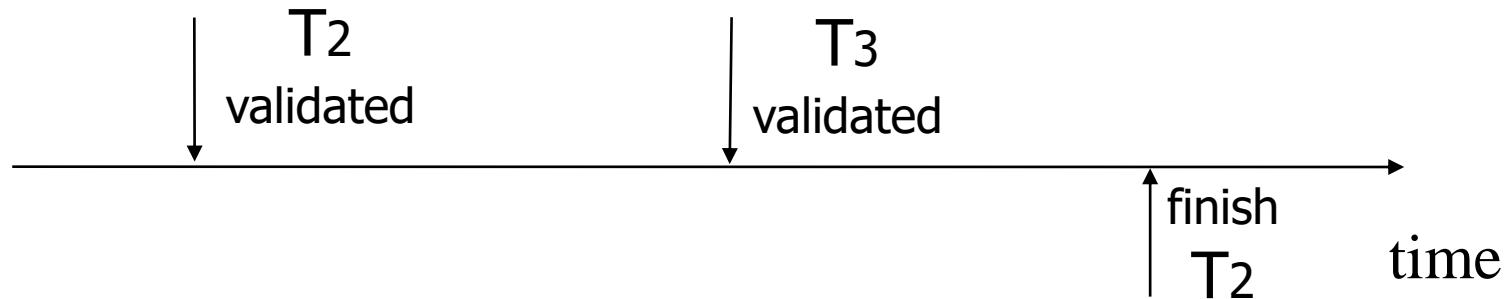
Another thing validation must prevent:

$$RS(T_2) = \{A\}$$

$$WS(T_2) = \{D, E\}$$

$$RS(T_3) = \{A, B\}$$

$$WS(T_3) = \{C, D\}$$



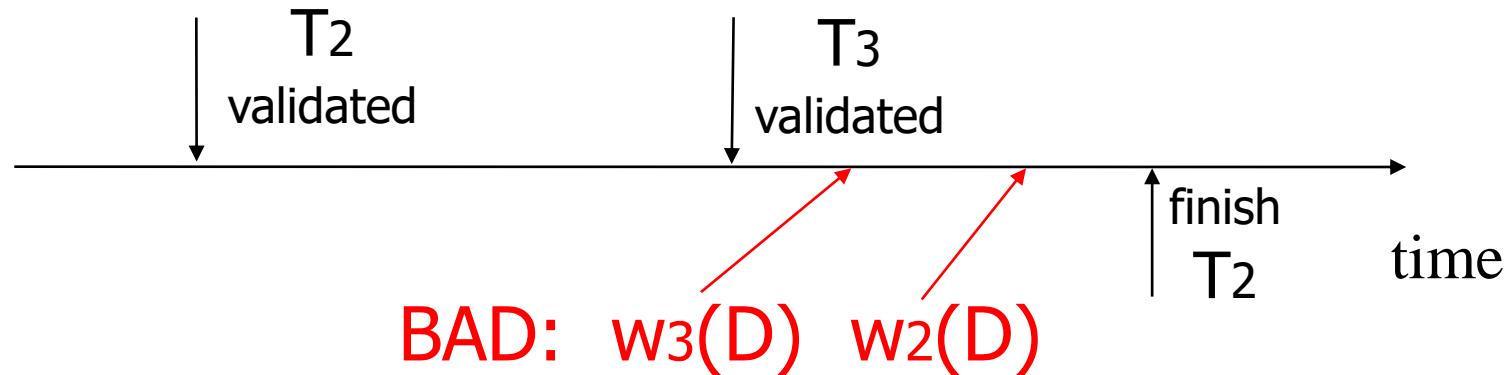
Another thing validation must prevent:

$$RS(T_2) = \{A\}$$

$$WS(T_2) = \{D, E\}$$

$$RS(T_3) = \{A, B\}$$

$$WS(T_3) = \{C, D\}$$



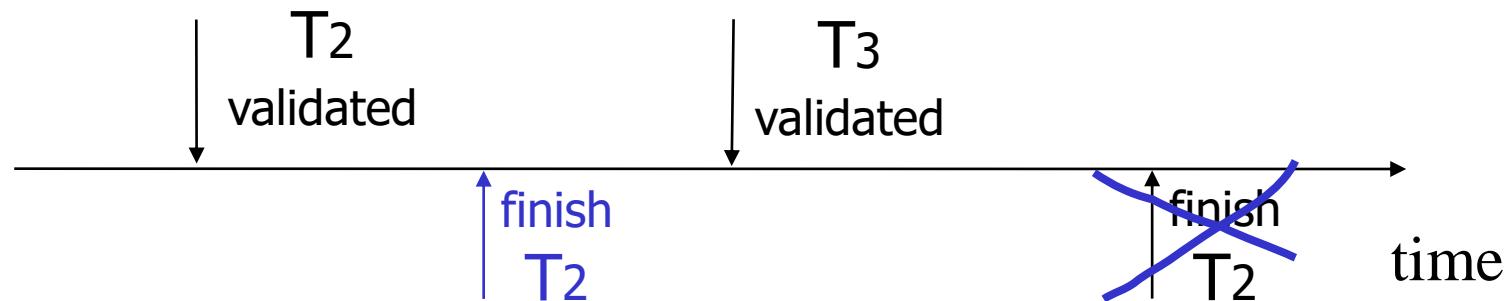
Another thing validation must prevent:

$$RS(T_2) = \{A\}$$

$$WS(T_2) = \{D, E\}$$

$$RS(T_3) = \{A, B\}$$

$$WS(T_3) = \{C, D\}$$



Validation rules for T_j :

(1) When T_j starts phase 1:

$\text{ignore}(T_j) \leftarrow \text{FIN}$

(2) at T_j Validation:

if check (T_j) then

[$\text{VAL} \leftarrow \text{VAL} \cup \{T_j\}$;

do write phase;

$\text{FIN} \leftarrow \text{FIN} \cup \{T_j\}$]

Check (T_j):

For $T_i \in \text{VAL} - \text{IGNORE } (T_j)$ DO
 IF [$\text{WS}(T_i) \cap \text{RS}(T_j) \neq \emptyset$ OR
 $T_i \notin \text{FIN}$] THEN RETURN false;
 RETURN true;

Check (T_j):

For $T_i \in \text{VAL} - \text{IGNORE}(T_j)$ DO
 IF [$\text{WS}(T_i) \cap \text{RS}(T_j) \neq \emptyset$ OR
 $T_i \notin \text{FIN}$] THEN RETURN false;
 RETURN true;

Is this check too restrictive ?

Improving Check(T_j)

For $T_i \in \text{VAL} - \text{IGNORE}(T_j)$ DO

IF [$\text{WS}(T_i) \cap \text{RS}(T_j) \neq \emptyset$ OR

$(T_i \notin \text{FIN} \text{ AND } \text{WS}(T_i) \cap \text{WS}(T_j) \neq \emptyset)$]

THEN RETURN false;

RETURN true;

Exercise:

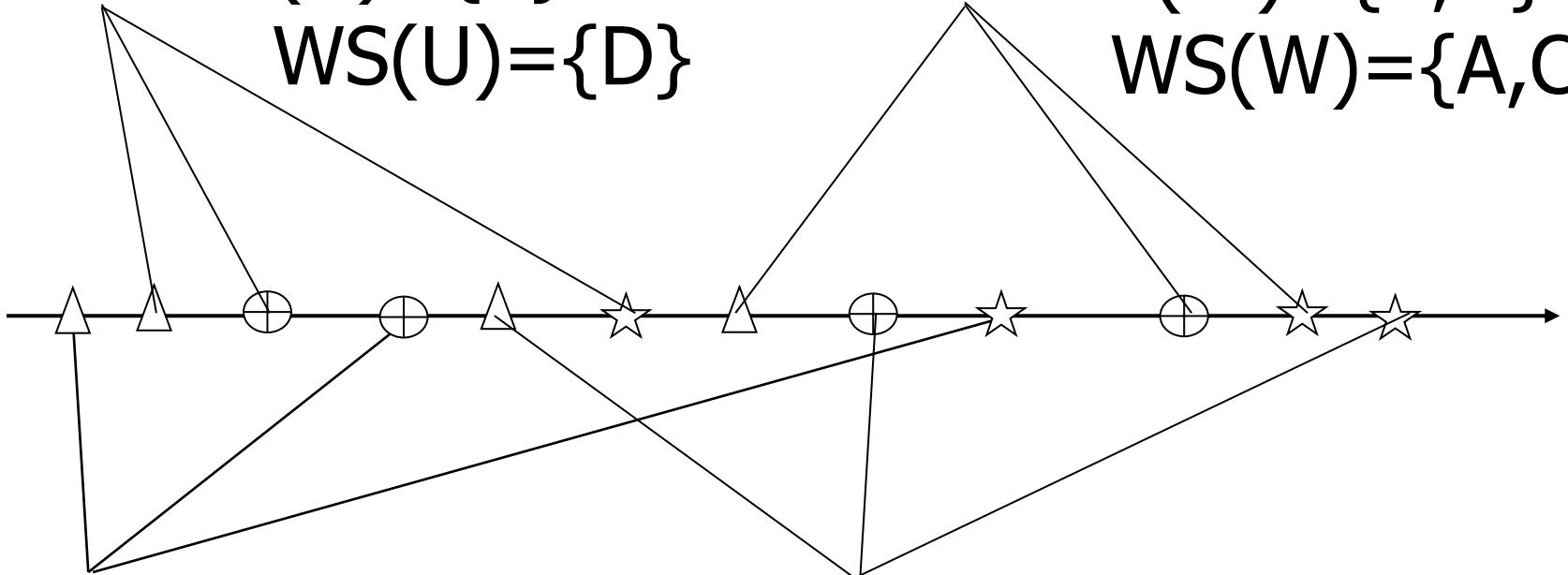


U: $RS(U)=\{B\}$

$WS(U)=\{D\}$

W: $RS(W)=\{A,D\}$

$WS(W)=\{A,C\}$



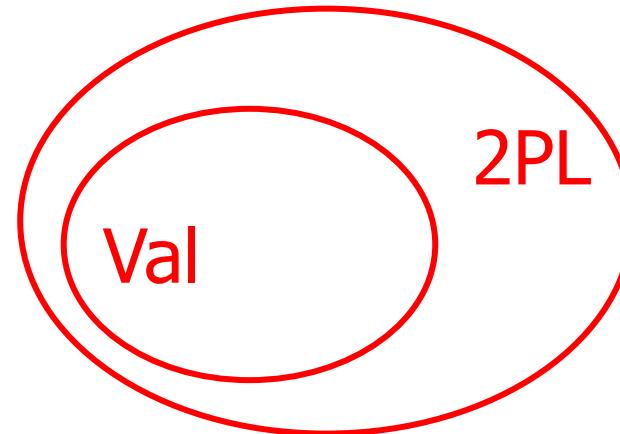
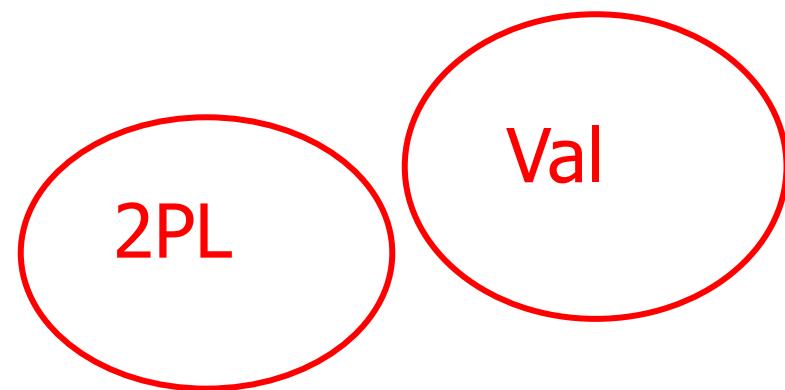
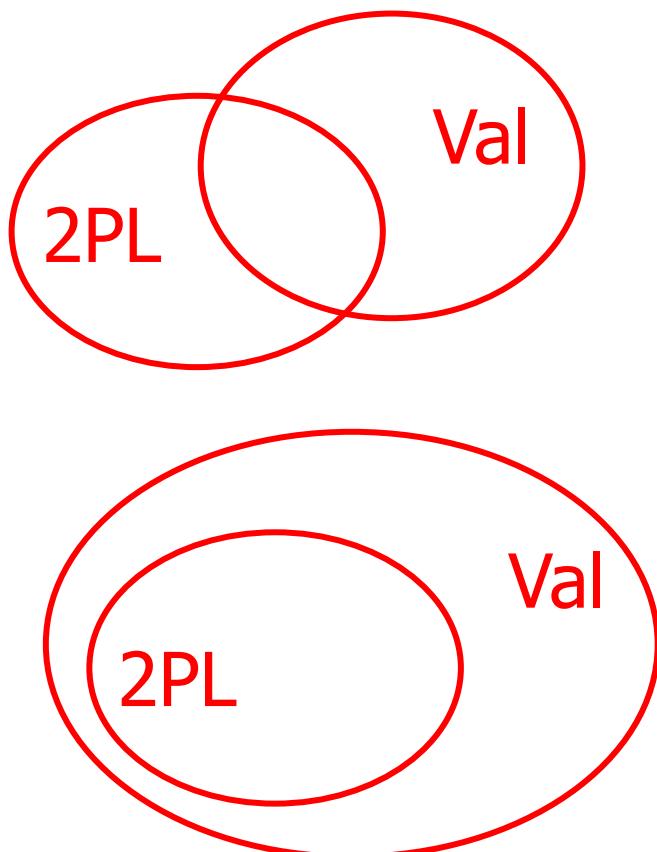
T: $RS(T)=\{A,B\}$

$WS(T)=\{A,C\}$

V: $RS(V)=\{B\}$

$WS(V)=\{D,E\}$

Is Validation = 2PL?



S2: $w2(y) \ w1(x) \ w2(x)$

- Achievable with 2PL?
- Achievable with validation?

S2: w2(y) w1(x) w2(x)

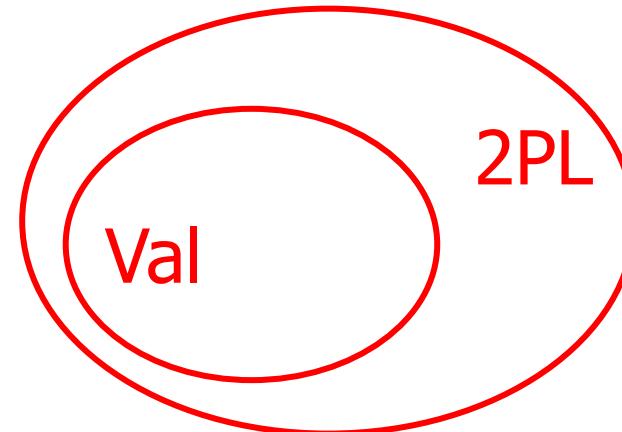
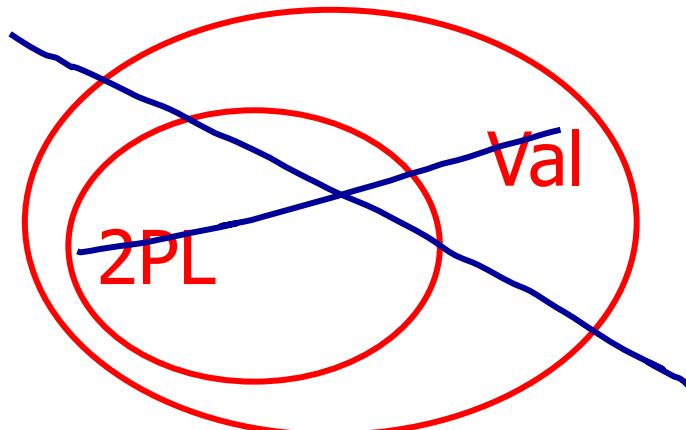
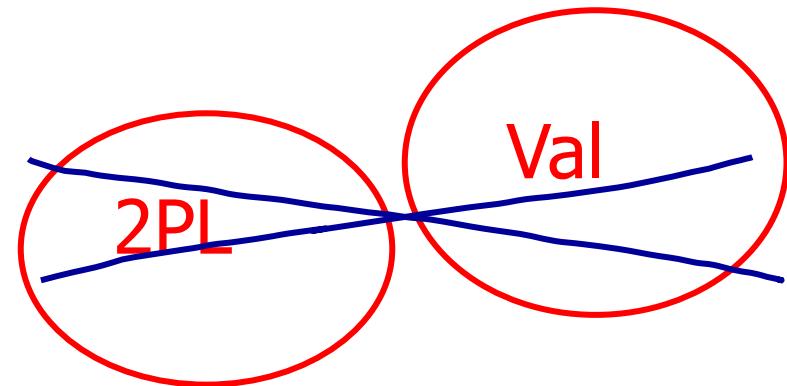
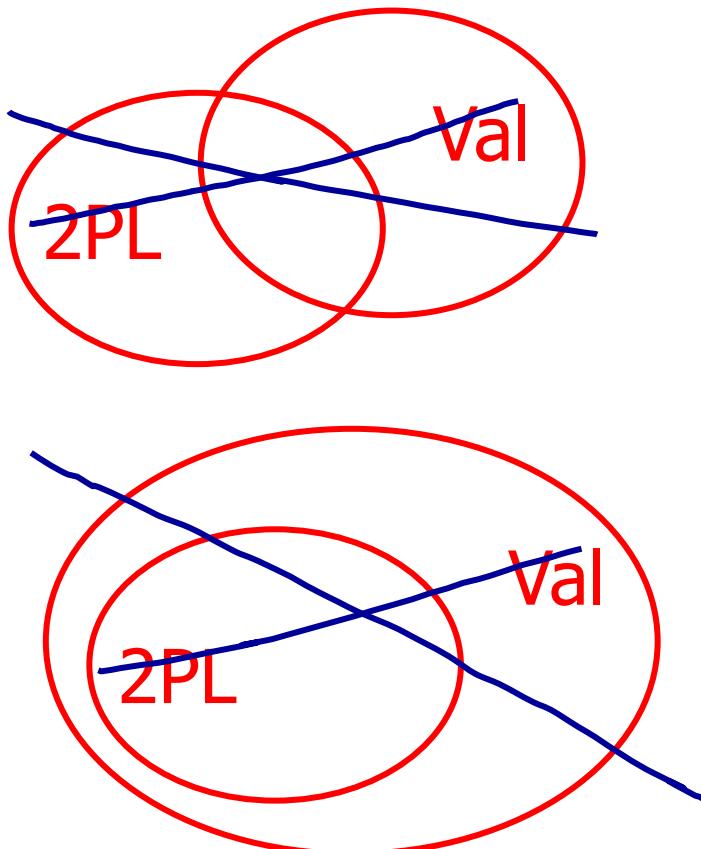
- S2 can be achieved with 2PL:
 $I2(y) w2(y) I1(x) w1(x) u1(x) I2(x) w2(x) u2(y) u2(x)$
- S2 cannot be achieved by validation:
The validation point of T2, val2 must occur before w2(y) since transactions do not write to the database until after validation. Because of the conflict on x, val1 < val2, so we must have something like
S2: val1 val2 w2(y) w1(x) w2(x)
With the validation protocol, the writes of T2 should not start until T1 is all done with its writes, which is not the case.

Validation subset of 2PL?

- Possible proof (Check!):
 - Let S be validation schedule
 - For each T in S insert lock/unlocks, get S' :
 - At T start: request read locks for all of $RS(T)$
 - At T validation: request write locks for $WS(T)$; release read locks for read-only objects
 - At T end: release all write locks
 - Clearly transactions well-formed and 2PL
 - Must show S' is legal (next page)

- Say S' not legal (due to w-r conflict):
 $S': \dots l1(x) \quad w2(x) \quad r1(x) \quad val1 \quad u1(x) \dots$
 - At $val1$: $T2$ not in $Ignore(T1)$; $T2$ in VAL
 - $T1$ does not validate: $WS(T2) \cap RS(T1) \neq \emptyset$
 - contradiction!
- Say S' not legal (due to w-w conflict):
 $S': \dots val1 \quad l1(x) \quad w2(x) \quad w1(x) \quad u1(x) \dots$
 - Say $T2$ validates first (proof similar if $T1$ validates first)
 - At $val1$: $T2$ not in $Ignore(T1)$; $T2$ in VAL
 - $T1$ does not validate:
 $T2 \notin FIN \text{ AND } WS(T1) \cap WS(T2) \neq \emptyset$
 - contradiction!

Conclusion: Validation subset 2PL



Validation (also called optimistic concurrency control) is useful in some cases:

- Conflicts rare
- System resources plentiful
- Have real time constraints

Summary

Have studied C.C. mechanisms used in practice

- 2 PL
- Multiple granularity
- Tree (index) protocols
- Validation

Oracle Transactions

A multiuser database must provide the following:

- The assurance that users can access data at the same time (**data concurrency**)
- The assurance that each user sees a consistent view of the data (**data consistency**), including visible changes made by the **user's own transactions** and **committed transactions** of other users

To describe consistent transaction behavior when transactions run concurrently, database researchers have defined a transaction isolation model called **serializability**. A serializable transaction operates in an environment that makes it appear as if no other users were modifying data in the database.

While this degree of isolation between transactions is generally desirable, running many applications in serializable mode can seriously compromise application throughput. Complete isolation of concurrently running transactions could mean that one transaction cannot perform an insertion into a table being queried by another transaction. In short, real-world considerations usually require a compromise between perfect transaction isolation and performance.

Oracle Database maintains data consistency by using a **multiversion consistency model** and **various types of locks** and transactions. In this way, the database can present a view of data to multiple concurrent users, with each view consistent to a point in time. Because different versions of data blocks can exist simultaneously, transactions can read the version of data committed at the point in time required by a **query** and return results that are **consistent to a single point in time**.

Oracle Database **never permits a dirty read**, which occurs when a transaction reads uncommitted data in another transaction.

To illustrate the problem with dirty reads, suppose one transaction updates a column value without committing. A second transaction reads the updated and dirty (uncommitted) value. The first session rolls back the transaction so that the column has its old value, but the second transaction proceeds using the updated value, corrupting the database. Dirty reads compromise **data integrity**, violate foreign keys, and ignore unique constraints.

Oracle Database always enforces **statement-level read consistency**, which guarantees that data returned by a single query is committed and **consistent for a single point in time**. The point in time to which a single SQL statement is consistent depends on the transaction isolation level and the nature of the query:

In the **read committed** isolation level, this point is the time at which the **statement** was opened.

In a **serializable** or **read-only** transaction, this point is the time the **transaction** began.

Oracle Database can also provide read consistency to all queries in a transaction, known as **transaction-level read consistency**. In this case, each statement in a transaction sees data from the *same* point in time, which is the time at which the transaction began.

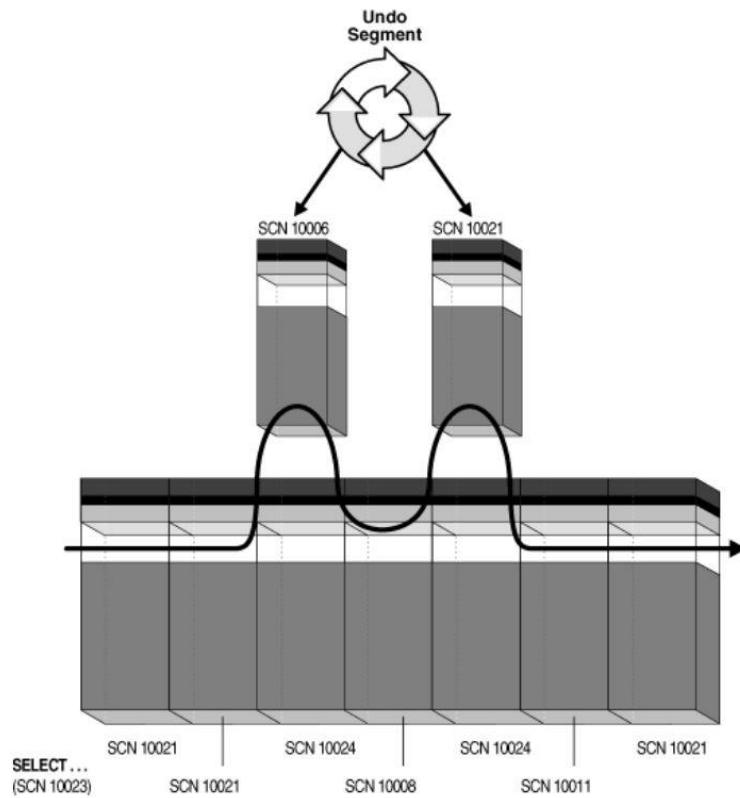
Queries made by a serializable transaction see changes made by the transaction itself. For example, a transaction that updates employees and then queries employees will see the updates. Transaction-level read consistency produces repeatable reads and does not expose a query to phantom reads.

To manage the multiversion read consistency model, the database must create a read-consistent set of data when a table is simultaneously queried and updated. Oracle Database **achieves read consistency through undo data**.

Whenever a user modifies data, Oracle Database **creates undo entries**, which it writes to undo segments. The undo segments contain the **old values of data** that have been changed by uncommitted or recently committed transactions. Thus, multiple versions of the same data, all at different points in time, can exist in the database. The database can use snapshots of data at different points in time to provide read-consistent views of the data and enable nonblocking queries.

As the database retrieves data blocks on behalf of a query, the database ensures that the data in each block reflects the contents of the block when the query began.

The database uses an **internal ordering mechanism** called an **SCN** to guarantee the order of transactions. As the **SELECT** statement enters the execution phase, the database determines the SCN recorded at the time the query began executing.



Blocks with **SCNs after 10023** indicate changed data, as shown by the two blocks with SCN 10024. The **SELECT** statement requires a version of the block that is consistent with committed changes. The database copies current data blocks to a new buffer and **applies undo data to reconstruct previous versions** of the blocks. These reconstructed data blocks are called *consistent read (CR) clones*.

The database creates two CR clones: one block consistent to SCN 10006 and the other block consistent to SCN 10021. The database returns the reconstructed data for the query.

The **block header** of every segment block contains an **interested transaction list (ITL)**. The database uses the ITL to determine whether a transaction was uncommitted when the database began modifying the block. Entries in the ITL describe which transactions have rows locked and **which rows in the block contain committed and uncommitted changes**. The ITL points to the transaction table in the undo segment, which provides information about the timing of changes made to the database.

In a sense, the block header contains a recent history of transactions that affected each row in the block. The **INITTRANS** parameter of the **CREATE TABLE** and **ALTER TABLE** statements controls the amount of transaction history that is kept.

The **SQL standard**, which has been adopted by ANSI, **defines four levels of transaction isolation**. These levels have differing degrees of impact on transaction processing throughput.

These isolation levels are defined in terms of phenomena that must be prevented between concurrently executing transactions. The **preventable phenomena** are:

1. Dirty reads

A transaction reads data that has been written by another transaction that has not been committed yet.

2. Nonrepeatable (fuzzy) reads

A transaction **rereads data** it has previously read and finds that another committed transaction has modified or deleted the data. For example, a user queries a row and then later queries the same row, only to discover that the data has changed.

3. Phantom reads

A transaction reruns a query returning a set of rows that satisfies a search condition and finds that another committed transaction has **inserted additional rows** that satisfy the condition.

For example, a transaction queries the number of employees. Five minutes later it performs the same query, but now the number has increased by one because another user inserted a record for a new hire. More data satisfies the query criteria than before, but unlike in a fuzzy read the previously read data is unchanged.

The SQL standard defines **four levels** of isolation in terms of the phenomena that a transaction running at a particular isolation level is permitted to experience.

Read uncommitted (possible: 1,2,3)

Read committed (possible: 2,3)

Repeatable read (possible: 3)

Serializable (possible: none)

Read Committed Isolation Level

In the **read committed isolation level**, which is the **default**, every query executed by a transaction **sees only data committed before the query** - not the transaction - **began**.

A query in a read committed transaction avoids reading data that commits while the query is in progress. For example, if a query is halfway through a scan of a million-row table, and if a different transaction commits an update to row 950,000, then the query does not see this change when it reads row 950,000. However, because the database does not prevent other transactions from modifying data read by a query, other transactions may change data *between* query executions. Thus, a transaction that runs the same query twice may experience fuzzy reads and **phantoms**.

In a read committed transaction, a **conflicting write** occurs when the transaction attempts to change a row updated by an uncommitted concurrent transaction, sometimes called a *blocking transaction*. The read committed transaction **waits for the blocking transaction** to end and release its row lock.

If the blocking transaction **rolls back**, then the waiting transaction proceeds to change the previously locked row as if the other transaction never existed.

If the blocking transaction **commits** and releases its locks, then the waiting transaction proceeds with its intended **update to the newly changed row**.

Serializable Isolation Level

In the **serializable isolation level**, a transaction **sees only changes committed at the time the transaction** - not the query - **began** and changes made by the transaction itself. A serializable transaction operates in an environment that makes it appear as if no other users were modifying data in the database.

Oracle Database permits a serializable transaction to modify a row only if changes to the row made by other transactions were *already* committed when the serializable transaction began. The database generates an error when a serializable transaction tries to update or delete data changed by a different transaction that committed *after* the serializable transaction began:

ORA-08177: **Cannot serialize access** for this transaction

The **read-only isolation level** is similar to the serializable isolation level, but read-only transactions do not permit data to be modified in the transaction.

Setting the isolation level:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE | READ ONLY| READ COMMITTED
ALTER SESSION SET ISOLATION LEVEL SERIALIZABLE | READ COMMITTED
ALTER SYSTEM ...
```

	T1	T2
1.	read(X)	
2.		write(X)
3.		commit
4.	read(X)	

Question: Can T1 see the new X value in step 4?

Example:

```
CREATE TABLE tr_proba(sorsz NUMBER(4), szam NUMBER, szoveg VARCHAR2(40));
INSERT INTO tr_proba VALUES(1, 10, 'First row');
INSERT INTO tr_proba VALUES(2, 20, 'Second row');
```

```
1. session                                2. session

SET AUTOCOMMIT OFF
----->      SET AUTOCOMMIT OFF
SELECT * FROM tr_proba;
----->      SELECT * FROM tr_proba;
UPDATE tr_proba
SET szam=szam+1
WHERE sorsz=1;
----->      SELECT * FROM tr_proba;
COMMIT;
----->      SELECT * FROM tr_proba; (can see new value)

----->      COMMIT;
----->      SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM tr_proba;
----->      SELECT * FROM tr_proba;
UPDATE tr_proba
SET szam=szam+1
WHERE sorsz=1;
----->      SELECT * FROM tr_proba;
COMMIT;
----->      SELECT * FROM tr_proba; (cannot see new value)
```

Locks in Oracle

In general, the database uses two types of locks: **exclusive** locks and **share** locks. Only one exclusive lock can be obtained on a resource such as a row or a table, but many share locks can be obtained on a single resource. For tables there are **other lock modes** as well.

(V\$LOCK table lmode column)

(RS -> 2)	LOCK TABLE <tab> IN ROW SHARE MODE
(RX -> 3)	LOCK TABLE <tab> IN ROW EXCLUSIVE MODE
(S -> 4)	LOCK TABLE <tab> IN SHARE MODE

(SRX -> 5) LOCK TABLE <tab> IN SHARE ROW EXCLUSIVE MODE
(X ->6) LOCK TABLE <tab> IN EXCLUSIVE MODE

Queries about locks held by sessions:

```
SELECT * FROM v$session WHERE username=USER;
```

Session id (sid) of the current query can be queried with the following:

```
SELECT sys_context('userenv', 'sid') FROM dual;
```

Which session hold a lock and since when (CTIME)?

```
SELECT se.sid, se.username, lo.type, lo.lmode, lo.ctime
FROM v$lock lo, v$session se
WHERE se.sid = lo.sid AND username = 'NIKOVITS';
```

SID	USERNAME	TYPE	LMODE	CTIME
305	NIKOVITS	TM	3	18
305	NIKOVITS	TX	6	18

TM means table lock

TX means row lock

See V\$LOCK_TYPE

Which session waits for a lock (REQUEST > 0), which is a blocking session (BLOCK = 1), and for what time does it wait or block (CTIME)?

```
SELECT se.sid, se.username, lo.type, lo.lmode,
       lo.request, lo.ctime, block
FROM v$lock lo, v$session se
WHERE se.sid = lo.sid AND username = 'NIKOVITS';
```

SID	USERNAME	TY	LMODE	REQUEST	CTIME	BLOCK
16	NIKOVITS	TM	3	0	4499	0
16	NIKOVITS	TX	6	0	4499	1
29	NIKOVITS	TX	0	6	186	0
29	NIKOVITS	TM	3	0	186	0

Which objects are locked currently?

```
SELECT lo.oracle_username, lo.session_id, lo.locked_mode,
       db.object_name, db.object_type
FROM v$locked_object lo, dba_objects db
WHERE lo.object_id = db.object_id and oracle_username = 'NIKOVITS';
```

ORACLE_USERNAME	SESSION_ID	LOCKED_MODE	OBJECT_NAME	OBJECT_TYPE
NIKOVITS	16	3	TR_PROBA	TABLE