Buffer

- ❑ DBMS stores information persistently on (“hard”) disks.
- ❑ Unit of transfer main-memory/disk: disk blocks or pages.
- ❑ Timing:
  - ✩ 2- 20 msec for random data block (bad seek time)
  - ✩ If blocks are sequentially on disk, only +1ms per block
  - ✩ Compare main memory access: in nanoseconds
- ❑ Basic operations (READ/WRITE from/to disk)
- ❑ Why disks?
  - ✩ Cheaper than Main Memory
  - ✩ Higher Capacity
  - ✩ Main Memory is volatile

- ❑ When loading a page from disk:
  - ✩ Replacement frame must have “pin counter” of 0
- ❑ When requesting a page that is in the buffer
  - ✩ Increment pin counter
- ❑ After operation has finished
  - ✩ Decrement pin counter
  - ✩ Set dirty bit if page has been modified:
- ❑ Frame is chosen for replacement by a replacement policy:
  - ✩ Only unpinned page can be chosen (pin count = 0)
  - ✩ Least-recently-used (LRU), Clock, MRU etc.

If requested is not in pool:
- ✩ If there is an empty frame, use it
- ✩ Else choose an empty frame for replacement. If the frame is dirty (page was modified), write it to disk
- ✩ Read requested page into chosen frame

Buffer management in DBMS requires ability to:
- ✩ **pin a page** in buffer pool, **force a page to disk** (important for implementing CC & recovery),
- ✩ adjust **replacement policy**, and **pre-fetch pages** based on access patterns in typical DB operations.

Indexing

COST MODEL

Measure performances by simplifying the parameters (IO focused):
- ✩ only consider disk reads (ignore writes)
- ✩ only consider number of I/Os and not the individual time for each read (ignores page pre-fetch)
- ✩ Average-case analysis; based on several simplistic assumptions.

HEAP FILES

- ✩ Linked, unordered list of all pages of the file
- ✩ Is it good for:
  - ● scan retrieving all records (SELECT *)?
    - ▲ yes, you have to retrieve all pages anyway
  - ● equality search on primary key
    - ▲ not great: have to read on avg half the pages for 1 record
  - ● range search or equality search on non-primary key
    - ▲ not great, all pages need to be read
  - ● insert
    - ▲ yes, can insert anywhere
  - ● delete/update
    - ▲ depends on where

SORTED FILES

- ✩ Records are ordered according to one or more attributes of the relation
- ✩ Is it good for:
  - ● scan retrieving all records (SELECT *)?
    - ▲ yes, you have to retrieve all pages anyway
  - ● equality search on sort attribute
    - ▲ good: find first qualifying page with binary search (log2)
  - ● range search on sort attribute
    - ▲ good: find first qualifying page with binary search (log2): adjacent pages might have additional matching records
  - ● insert
    - ▲ not good: have to find proper page; overflow possible
  - ● delete/update
    - ▲ finding tuples is fast.
    - ▲ BUT updading might lead to restructuring of pages.
  - ● Sorted output: (ORDER BY)
    - ▲ good if on sorted attribute

BUILDING INDEXES

Build an index for any attribute (collection of attributes) that is frequently used in queries:
- ✩ Additional information that helps finding specific tuples faster
- ✩ We call the collection of attributes over which the index is built the search key attributes for the index.
- ✩ Any subset of the attributes of a relation can be the search key for an index on the relation.
- ✩ Search key is not the same as primary key / key candidate

```
CREATE INDEX ind1 ON Skaters(sid);
DROP INDEX ind1;
```

B+ TREES

- ❑ height-balanced: Each path from root to tree has the same height
- ❑ F = fanout = number of children for each node (~ number of index entries stored in node)
- ❑ N = # leaf pages
- ❑ Insert/delete at log F N cost;
- ❑ Minimum 50% occupancy (except for root).

INSERT

- ❑ Find correct leaf L.
- ❑ Put data entry onto L.
  - ✩ If L has enough space, done !
  - ✩ must split L (into L and a new node L2)
    - ● Redistribute entries evenly, copy up middle key.
    - ● Insert index entry pointing to L2 into parent of L .
- ❑ This can happen recursively
  - ✩ To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- ❑ Splits “grow” tree; root split increases height
  - ✩ Tree growth: gets **wider** or **one level taller at top**.

INDIRECT INDEXING

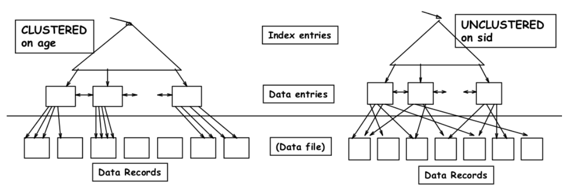
(10, rid1), (10, rid2), ... vs. (10, rid1, rid2, rid3), ...
- ✩ first requires more space (search key repeated)
- ✩ second has variable length data entries
- ✩ second can have large data entries that span a page

DIRECT INDEXING

Instead of data-entries in index leaves containing rids, they could contain the entire tuple. This is kind of a sorted file with an index on top.
- ✩ data-entry = tuple
- ✩ no extra data pages

CLASSIFICATION

<u>Primary vs. Secondary</u>
Primary Index if contains primary key. Unique Index.
<u>Clustered vs. Unclustered</u>
A file can be clustered on at most one search key.



INDEX ON MULTIPLE ATTRIBUTES

- ✩ CREATE INDEX ind1 ON Skaters(age,rating);
- ✩ Order is important:
  - ● Here data entries are first ordered by age
  - ● Skaters with the same age are then ordered by rating
- ✩ Supports:
  - ● SELECT * FROM Skaters WHERE age = 20;
  - ● SELECT * FROM Skaters WHERE age = 20 AND rating < 5;
- ✩ Does not support
  - ● SELECT * FROM Skaters WHERE rating < 5;

B+ TREES SIZE

- ✩ The average number of rids per data entry
  - ● Number of tuples / different values (if uniform) (Example 200,000/20,000 = 10)
- ✩ The average length per data entry:
  - ● Key value + #rids * size of rid (Example: 6 + 10*10 = 106)
- ✩ The average number of data entries per leaf page:
  - ● Fill-rate * page-size / length of data entry
  - ● Example: 0.75*4000 / 106 = 28 entries per page
- ✩ The estimated number of leaf pages:
  - ● Number of entries = number of different values / #entries per page
  - ● Example 20000 / 28 = 715
- ✩ Number of entries intermediate page:
  - ● Fill-rate * page-size /length of index entry
  - ● Min fill-rate: 0.5, max fill rate: 1
  - ● Example: 0.5 * 4000 / 14 = 143 entries ; 1* 4000/14 = 285 entries
- ✩ Height is 3: the root has between three and four children
  - ● Three children: each child has around 715/3 = 238 entries
  - ● Four children: each child has around 715/4 = 179 entrie

B+ TREE COST

- ✩ Relation R(A,B,C,D,E,F)
- ✩ A and B are int (each 6 Bytes), C-F is char[40] (160 Bytes)
  - ● Size of tuple: 172 Bytes
- ✩ 200,000 tuples

Each data page has 4 K and is around 80% full
  - ● 200,000*172/(0.8*4000) = 10750 pages
- ✩ Values of B are within [0;19999] uniform distribution
- ✩ Non-clustered B-tree for attribute B, indirect indexing (2)
- ✩ An index page has 4K and intermediate pages filled [50%-100%]
- ✩ The size of an rid = 10 Bytes
- ✩ The size of a pointer in intermediate pages: 8 Bytes
- ✩ Index entry size in root and intermediate pages:
  - ● size(key)+size(pointer) = 6 Bytes + 8 Bytes = 14 Bytes

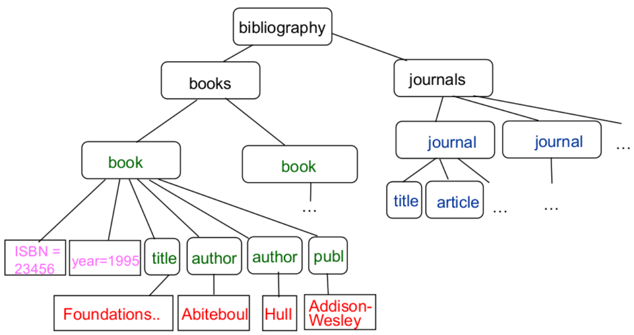
B+ TREES STATS

Height 4: 133 4 = 312,900,721 records
Height 3: 133 3 = 2,352,637 records
Height 2: 133 2 = 17,689 records
Typical order d of inner nodes: 100 (I.e., an inner node has between 100 and 200 index entries)
- ✩ Typical fill-factor: 67%
- ✩ average fanout = 133

Can often hold up top levels in buffer pool:
- ✩ Level 1 (root) = 1 page = 4 Kbytes
- ✩ Level 2 = 133 pages = 0.5 Mbyte
- ✩ Level 3 = 17,689 pages = 70 MBytes

XML

```
<bibliography>
    <books>
        <book ISBN="23456" year="1995">
            <title> Foundations ... </title>
            <author> Hull </author>
            <author> Abiteboul </author>
            <publ> Addison Wesley </publ>
        </book>
        <book> ...</book>
    </books>
    <journals>
        <journal>
            <title> ... </title>
            <article> ... </article>
            ...
        </journal>
        <journal> ... </journal>
    </journal>
</bibliography>
```




```
<DiscoverTheWorld>
        <tour TourId="1">
                <type> Brazil junge </type>
                <start-date> 16-April </start-date>
                <duration> 14 </duration>
                <price> 2229 </price>
        </tour>
        <tour TourId="2">
                <type> Brazil junge </type>
                <start-date> 30-April </start-date>
                <duration> 21 </duration>
                <price> 2999 </price>
        </tour>
        <tour TourId="3">
                <type> Kenia safari </type>
                <start-date> 30-April </start-date>
                <duration> 21 </duration>
                <price> 3229 </price>
        </tour>
        <reservation ResId="541" TourId="1">
                <cname> Bettina Kemme </cname>
                <caddress> Montreal </caddress>
                <cost> 2579 </cost>
                <special price="5"> vegetarian </special>
                <special price="20"> single </special>
        </reservation>
        <reservation ResId="542" TourId="2">
                <cname> Your Name </cname>
                <caddress> Your Address </caddress>
                <cost> 3105 </cost>
                <special price="5"> vegetarian </special>
        </reservation>
</DiscoverTheWorld>
```

```
<!DOCTYPE DiscoverTheWorld [
<!ELEMENT DiscoverTheWorld (tour*,reservation*)>
<!ELEMENT tour (type, start-date, duration, price) >
<!ELEMENT reservation (cname, caddress, cost, special*)>
<!ATTLIST tour TourId ID #REQUIRED >
<!ATTLIST reservation ResID ID #REQUIRED
TourID IDREF #REQUIRED>
<!ELEMENT type (#PCDATA) >
<!ELEMENT start-date (#PCDATA) >
<!ELEMENT duration (#PCDATA) >
<!ELEMENT price (#PCDATA) >
<!ELEMENT cname (#PCDATA) >
<!ELEMENT caddress (#PCDATA) >
<!ELEMENT cost (#PCDATA) >
<!ELEMENT special (#PCDATA) >
<!ATTLIST special price CDATA #REQUIRED>
]>
```

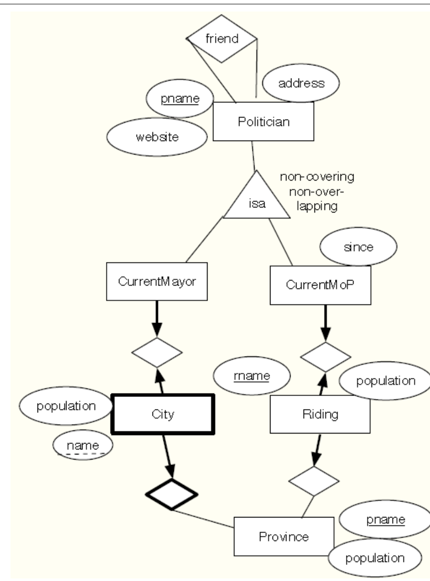
```
Return the names of all current mayors.
FOR $p IN /Politics/Politician[count(CurrentMayor) = 1]
RETURN $p/@pname
```

```
Return the province with the largest number of cities
LET $m := max(/Politics/Province/COUNT(City))
LET $c := /Politics/Province/[COUNT(City) = $m]
RETURN $c
```

```
List the name of all Mps in Quebec
OR $r in /Politics/Province[@pname = "Quebec"]/Riding
FOR $p in /Politics/Politician
WHERE $r/@rname = $p/CurrentMoP/@rname
RETURN $p/@pname
```

- DOM: Language neutral API with implementations in Java, C++,etc.
- Functionality
  - Construct a DOM tree from an XML document
  - Traverse and read a DOM tree
  - Construct an empty DOM tree
  - Modify an existing tree
  - Copy subtrees
- Nodes in DOM tree:
  - Document, element, attribute, text

```
<!DOCTYPE Politics [
<!ELEMENT Politics (Politician*, Province*)>
<!ELEMENT Politician ((CurrentMayor | CurrentMop)?, address?)>
<!ELEMENT CurrentMayor (since?)>
<!ELEMENT CurrentMoP (since?)>
<!ELEMENT Province (City+, Riding+, population?)>
<!ELEMENT City (population?)>
<!ELEMENT Riding (population?)>
<!ELEMENT address (#PCDATA) >
<!ELEMENT since (#PCDATA) >
<!ELEMENT population (#PCDATA) >
<!ATTLIST Politician pname ID REQUIRED
website CDATA IMPLIED
friends IDREFS IMPLIED>
<!ATTLIST CurrentMayor cityID IDREF REQUIRED>
<!ATTLIST CurrentMoP rname IDREF REQUIRED>
<!ATTLIST Province pname ID REQUIRED>
<!ATTLIST City cityID ID REQUIRED
cname CDATA REQUIRED>
<!ATTLIST Riding rname CDATA REQUIRED>
]>
```



DTD

```
<!DOCTYPE people [
<!ELEMENT people (person*)>
<!ELEMENT person (name*, (lastname|familyname)?)>
<!ATTLIST person PID ID #REQUIRED
  age CDATA #IMPLIED
  children IDREFS #IMPLIED
  mother IDREFS #IMPLIED
>
<!ELEMENT name (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT familyname (#PCDATA)>
]>
```

For and let in XPATH

```
Data types: PCDATA (parsed character data) or CDATA (unparsed)
Attributes
  ID unique identifier (similar to primary key)
  IDREF: reference to single ID
  IDREFS: space-seperated list of references
Values
  can give a default value
  #REQUIRED must exist
  #IMPLIED optional
Specified in an XML file with <!DOCTYPE name SYSTEM
"path/to/thing.dtd">
Can use regex style things too. * is 0 or more. + is 1 or more, (a
| b)? is one or the other
```

```
INSERT INTO MyXML(id, INFO) VALUES (1000,
  '<customerinfo cid="1000">
  <name>Kathy Jones</name>
  <addr country =Canada">
    <street>123 fake</street>
    <city>Ottawa</city>
    <prov-state>Ontario</prov-state>
    <pcode-zip>H0H 0H0</pcode-zip>
  </addr>
  </customerinfo>')
```

XPATH

```
-Basic Queries:   '/'        to navigate one path at a time
Example:/Bookstore/Book/Title
'//'      all paths following this
Example://Title
When wanting to access an attribute of an element use @
Example:/Bookstore/Book/data(@ISBN)
'|'  OR operator ONLY USED INSIDE CONDITIONS
Example:/Bookstore/Book|Magazine/Title
'='        can act like == like in Self-Join Queries below
Navigation accesses:
Example: parent::*  *::child  following-silbling::*
-Queries involving CONDITIONS
1condition:
Example:/Bookstore/Book[@Price<30]
Example:/Bookstore/Book/Authors/Author[2]
the 2nd author of each element
2conditions:
To write a condition, it needs to be inside '[...]' then followed
by the output that we are looking for
Example:/Bookstore/Book[@Price<30]/Title
Condition to find elements that contain other elements:
Example:/Bookstore/Book[Remark]/Title
```

Conditions && Conditions + Some output:
Example: Looking for a title that has one last name =Ullman and price<90
/Bookstore/Book[@Price<90 and Authors/Author/Last_Name="Ullman"]/Title
Conditions Inside Conditions + Some output:
Example:Looking for a title with author ="Jeffrey Ullman" and price<90
/Bookstore/Book[@Price<90 and Authors/Author/[Last_Name="Ullman" and First_Name="Jeffrey"]]/Title
Conditions && !Conditions + Some output:
Example:Looking for a title with author ="Ullman" and NOT author="Widom"
/Bookstore/Book[/Authors/Author/Last_Name="Ullman" and count(/Authors/Last_Name="Widom"=0]/Title
The condtion 'contains':
/Bookstore/Book[contains(Remark, "great")]/Title

<u>Self-Join Query</u>
Quering two instances of the database at one and joining them together.Trying to find the magazines wheres theres a book with the same title. Example:
doc("BookstoreQ.xml")/Bookstore/Magazine[Title=doc("BookstoreQ.xml")/Bookstore/Book/Title]
<u>Navigation Accesses</u>
The name() function returns the name of a tag or element
To find all elements whose parent is not "Bookstore" or "Book"
/Bookstore/*[name(parent::*)!="Bookstore" and name(parent::*)!="Book"]

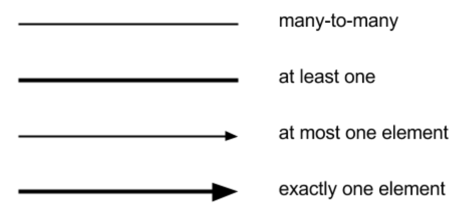
Queries Example

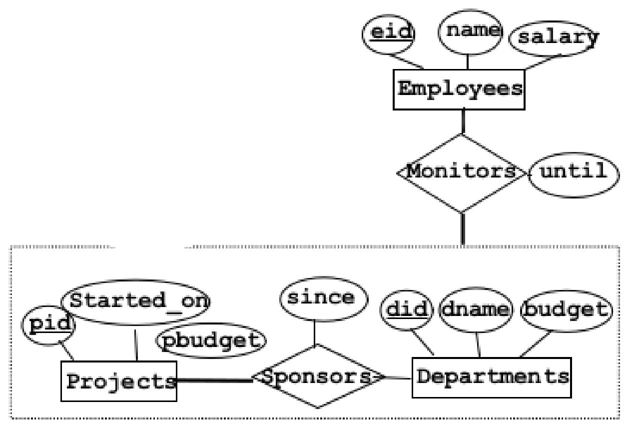
 /bibliography/book/author
  all author elements by root navigating through those elements
 /bibliography/book/@ISBN
  All ISBN attributes
 //title
  All title elements anywhere in the document
 /bibliography/*/title
  Titles of bibliography entries assuming that there could be books, journals, reports, etc...
 /bibliography/book[@year>1995]
  Returns books where the year > 1995
 /bibliography/book[author='FooBar']/@Year
  Returns the years of books written by FooBar
 /bibliography/book[count(author) <2]
/bibliography/book/author[position()=1]/name
  Position is the location of the node in the node set

For and let in XPATH

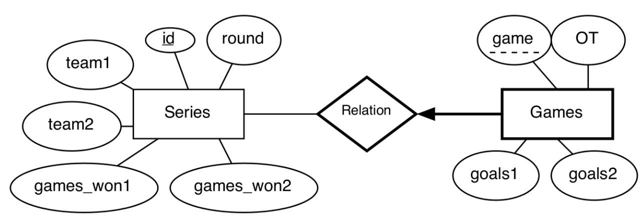
```
for $b in document("bib.xml")/bib/book
return <result> $b</result>

  <result><book>...</book></result>
  <result><book>...</book></result>
  <result><book>...</book></result>
  ...
  <result><book>...</book></result>

let $b in document("bib.xml")/bib/book
return <result> $b</result>

  <result>
    <book>...</book>
    <book>...</book>
    ..
    <book>...</book>
  </result>
```

ER Diagrams





SQL Queries samples



Return the ids of those series where at least one game was played with at least four goals difference. Return the ids in ascending order.

```
SELECT DISTINCT id
FROM games
WHERE goals1-goals2 > 3
OR
goals2-goals1 > 3
ORDER BY id
```

Return the rounds where at least one game of one series required overtime (OT equal to 1).

```
SELECT round
FROM games, series
WHERE games.id = series.id
AND games.ot = 1
```

Return the ids of series where no game was decided in overtime.

```
SELECT id
FROM series
WHERE id NOT IN (SELECT id FROM games WHERE OT = 1)
```

Return the ids of series that have the same score in every game.

```
SELECT g1.id
FROM
games g1
WHERE NOT EXISTS (SELECT *
FROM
games g2
WHERE g1.id = g2.id
AND
(g1.goals1 != g2.goals1 OR g1.goals2 != g2.goals2))
```

Return the series id, the game and the total number of goals of the game with the lowest number of total goals.

```
SELECT id, game, goals1+goals2
FROM games
WHERE (goals1+goals2) <= ALL (
SELECT goals1+goals2 FROM games)
```

Return for each series, the id and the number of games.

```
SELECT id, COUNT(*)
FROM
games
GROUP BY id
```

Return the average number of goals over all games

```
SELECT (avg(goals1+goals2)
FROM games
```

A typical query is to show the results for a given team. Write a query in SQL that produces the above result for ”myteam” – whatever this might be and how far the Playoffs are. Games should be presented in the order they were played. Remember that a team can be team1 in one series and team2 in another series.

```
SELECT s.team1, g.goals1, s.team2, g.goals2
FROM
series s, games g
WHERE g.id = s.id
AND (team1 = ’myteam’ OR team2 = ’myteam’)
ORDER BY g.id, game
```

Extend above query so that only the results for the last series for which at least one game was played, is shown.

```
SELECT s.team1, g.goals1, s.team2, g.goals2
FROM series s, games g
WHERE g.id = s.id
AND ( team1 = ’myteam’ OR team2 = ’myteam’)
AND s.id = (
  SELECT MAX(id) FROM series
  WHERE team1 = ’myteam’ OR team2 = ’myteam’)
ORDER BY g.id, game
```

Return the ids of series that don’t have seven games

```
SELECT id
FROM games
GROUP BY id
HAVING COUNT(*) < 7
```

Return the ids of series where at least two games were decided in overtime.

```
SELECT g.id
FROM games g
WHERE g.OT = 1
GROUP BY g.id
HAVING COUNT(*) > 1
```

Return the ids of series of the Montreal Canadiens that don't have seven games.

```
SELECT g.id FROM games g, series s
WHERE s.id = g.id
AND ( s.team1 = 'Montreal Canadiens'
  OR s.team2 = 'Montreal Canadiens'
)
GROUP BY g.id
HAVING COUNT(*) < 7

SELECT id FROM series
WHERE ( team1 = 'Montreal Canadiens'
  OR team2 = 'Montreal Canadiens'
)
AND games_won1 + games_won2 < 7
```

Write a row-level trigger that guarantees that for each web-page only the latest 20 requests are recorded. That is, once there are 20 requests for a web-page and a new request for this web-page is entered into the Request table, then the entry with the smallest request id for that page should be removed from the table. Assume that an insert inserts only a single tuple.

```
CREATE TRIGGER tr
AFTER INSERT ON Request
REFERENCING NEW AS n
FOR EACH ROW
WHEN (20 < (SELECT COUNT(*) FROM Request WHERE page-id = n.page-id))
DELETE FROM Request
WHERE request-id = (SELECT MIN(request-id) FROM request where page-id = n.page-id)
```

TRIGGER TEMPLATE

```
CREATE TRIGGER trigger-name
/* triggering event
AFTER/BEFORE INSERT/DELETE ON table-name
REFERENCING NEW AS n
FOR EACH ROW
WHEN condition /* as in WHERE-clause
/* action */
SQL-statement
```

Employee (eid, ename, location, sid)
Employees have an id, a name, a location where they work, and a supervisor who is also an employee
(foreign key to Employees)
Project (pid, start-year, cid)
Projects have an id, a start year and a coordinator who is an employee (foreign key to Employees)
WorksFor (pid, eid, since)
Employees work for projects

1.Give the pids of all projects that started in 2010.
$\Pi_{pid}(\sigma_{start\text{-}year=2010}(Project))$
2. Give the pids of all projects that started either in 2007 or in 2008.
$\Pi_{pid}(\sigma_{start\text{-}year=2007 \wedge start\text{-}year=2008}(Project))$
3. Give the pids of all projects for which employees at the Montreal location work.
$\Pi_{pid}(WorksFor \bowtie \sigma_{location='Montreal'}(Employees))$
4. Give the pids of all projects for which employees at the Montreal and the Toronto locations work.
$\Pi_{pid}(WorksFor \bowtie \sigma_{location='Montreal'}(Employees))$
$\cap\ \Pi_{pid}(WorksFor \bowtie \sigma_{location='Montreal'}(Employees))$

SQL

2009 Q3

```
Tours(TourID, type, start_date, duration, price)
Specials(SpecID, name, price)
Reservations(ResID, TourID, cname, caddress, cost)
SpecialRes(ResID, SpecID)
```

Return ResID and cname for reservations that neither include the special "vegetarian" nor the special "single"

```
SELECT resID, cname FROM Reservations
WHERE resID NOT IN (
  SELECT resID FROM SpecialRes SR
    JOIN Special S ON SR.SpecID=S.SpecID
  WHERE name='vegetarian' OR name='single'

)
```

Return the resID and the number of specials of Reservations that have booked at least two specials.

```
SELECT resID, COUNT(*) FROM Specials
GROUP BY resID
HAVING COUNT(*) >= 2
```

Query Optimisation

REDUCTION FACTOR

Pushing down projections will not reduce the number of tuples but the SIZE of the intermediate results.

Result sizes: number of input tuples * reduction factor.
Selections with low reduction factor (high selectivity) should be executed as fast as possible (WHERE sid = 5, WHERE age = 6)

JOIN COST ESTIMATION

|Skaters Participates| = |Participates|

- Join attribute is primary key for Skaters
- Each Participates tuple matches exactly with one Skaters tuple

|Skaters × Participates| = |Participates| ∗ |Skaters|

- Cross product is always the product of individual relation sizes
- For other joins more difficult to estimate (Continues in next episode...)

JOIN cost on relation R1 and R2

BLOCK ORIENTED NESTED LOOP JOIN

Smaller relation fits in main memory+2extra buffer page:
cost = page(R1) + page(R2)
No relation fits in main memory(B join frame):

$$cost = page(R1) + \frac{page(R2) * page(R1)}{B-2}$$

INDEX NESTED LOOP JOIN

Index on the join column of one of the relation(R2):
cost = page(R1) + card(R1) ∗ cost_finding_index(R2)

If the join attribute is primary key in inner relation

SORT MERGE JOIN

- Sort P and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples
- Advance scan of P until current P-tuple >= current S tuple, then advance scan of S until current S-tuple >= current P tuple; do this until current P tuple = current S tuple.
- At this point, all P tuples with same value in Pi (current P group) and all S tuples with same value in Sj (current S group) match; output <p, s> for all pairs of such tuple 1s.

P is scanned once; each S group is scanned once per matching P tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

SORT MERGE JOIN

- Simple nested loop join: For each tuple in the outer relation P, we scan the entire inner relation S

cost = PartPages + CARD(P) ∗ SkaterPages = 1000 + 100000 ∗ 500

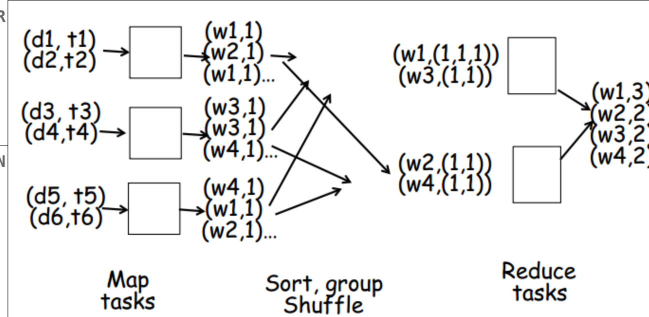
- Page-oriented Nested Loops join: For each page of P, get each page of S, and write out matching pairs of tuples <p, s>, where p is in P-page and s is in S-page.

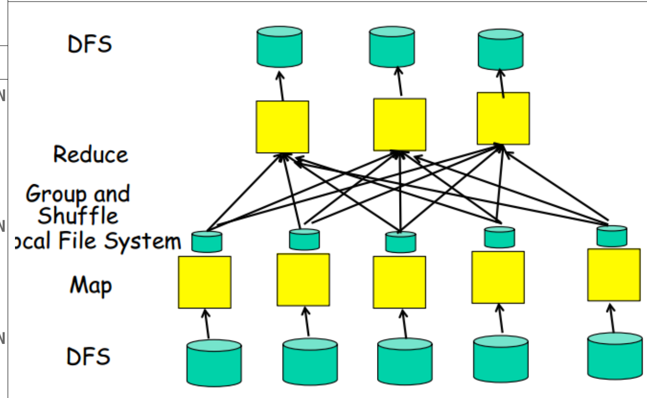
cost = PartPage + PartPage ∗ SkaterPage = 1000 + 1000 ∗ 500)

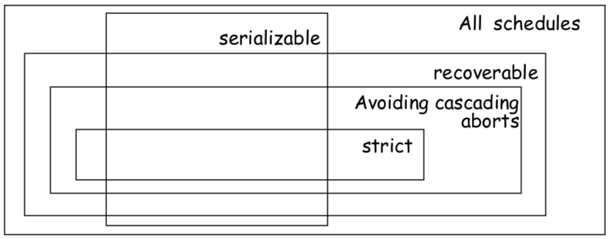
Big Data

SORT MERGE JOIN

```
Users = load 'users' as (name,age);
Fltrd = filter Users by age >= 18 and age <= 25;
Pages = load ' pages ' as (uname, url);
Jnd = join Fltrd by name, Pages by uname;
Grpd = group Jnd by url;
Smmd = foreach Grpd generate ($0), COUNT($1)
as clicks;
Srtd = order Smmd by clicks desc;
Top5 = limit Srtd 5;
store Top5 into ' top5sites '
```







Isolation Level\Anomaly	Dirty Read	Unrepeatable Read	Phantom
Read Uncommitted	maybe	maybe	maybe
Read Committed	no	maybe	maybe
Repeatable Reads	no	no	maybe
Serializable	no	no	no

NON SERIAL SCHEDULE

Serialisable: Dependency graph has no cycle
Recoverable Schedule: If transaction T_i reads a value written by transaction T_j then T_i commits only after T_j has commited.
Avoid Cascading Abort: A transaction reads only values written by comitted transactions.
Strict: A transaction only reads or overwrite values written by commited transactions.

Unrecoverable		Recoverable schedule with cascading abort		Recoverable schedule with commit		Avoids cascading		Non strict		Strict		Strict and serializable	
T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2
R(A)		R(A)		R(A)		R(A)		W(A)		W(A)		R(x)	
W(A)		W(A)		W(A)		W(A)			W(A)	abort			W(x)
	R(A)		R(A)		R(A)	abort		abort			W(A)		commit
	commit	abort		commit			R(A)		commit		commit	W(x)	
commit			abort		commit		commit					commit	

Unrepeatable read: If T1 read twice the same data item but T2 change its value between the first and the second
Dirty read: If T2 read from T1 before T1 commit.
Lost update: If T2 modify a data item modified by T1 without taking in account the value modified by T1.

FORCE FLUSH STRATEGY

All changes are flush to disk BEFORE commit
 Completed transaction need not action

 Active transaction might have partial changes on disk(Need undone)
 Append to log file log record before flushing
 At commit/abort append to log file commit/abort log record
 When recovering from crash: Scan log backward for each record if commited ignore otherwise install Before-Image of the record

NO FORCE FLUSH STRATEGY

 Changes might be flushed at any time(BEFORE/AFTER commit)

 Done transaction might have missing changes (must be redone)
 Active/Aborted transaction might have been flushed before crash(Must be undone)

 For each write(x) of a transaction T with x being on page P: Log record with before AND after image of x(Before so you can redo changes)
 Flush before-image to disk before flushing the P
 Flush after-image to disk before commit of T
 At commit/abort append commit/abort record to log file and flush

Let suppose we have a relation R (A, B, C, D, F) such that:
 A and B are int (6 byte)
 C-F are char [40] (10 byte per char).
 Tuple = 172 bytes. 200,000 tuples
 Each data page has 4000 bytes and is around 80% full
 B values are uniformly distributed
 Rid = 10 bytes
 Size of pointer in intermediate page = 8 bytes
 Index pages are 4K and between 50%-100% full

Goal	Formula	With this example
Number of pages	$\frac{number\ of\ tuples * tuple\ size}{fill\ rate * page\ size}$	$\frac{172 * 200000}{40000 * 0.80} = 10750$
Index entry size in root and intermediate pages	$size\ of\ key + size\ of\ pointer$	$6 + 8 = 14\ bytes$
Average number of rids per data entry	$\frac{\text{number of tuples}}{\text{different values (if uniform)}}$	$\frac{200{,}000}{20{,}000} = 10$
Average length per data entry	size of key + (number of rids * size of rid)	$6 + 10 * 10 = 106$
Average number of data entries per leaf page	$\frac{\text{fillrate} * \text{page size}}{\text{length of data entry}}$	$\frac{0.75 * 4000}{106} = 28\ entries\ per\ page$
Estimate number of leaf page	$\frac{\text{number of different values}}{\text{number of entrier per page}}$	$\frac{20{,}000}{28} = 715$
Number of entries in intermediate pages	$\frac{\text{fillrate} * \text{page size}}{\text{lenght of index enty}}$	$min = \frac{0.5 * 4000}{14} = 143, \max = \frac{1 * 4000}{14} = 285$
Height of tree	$(nb\ of\ entry\ in\ intermediate\ page)^{h-1} > nb\ of\ leaf\ page$	3