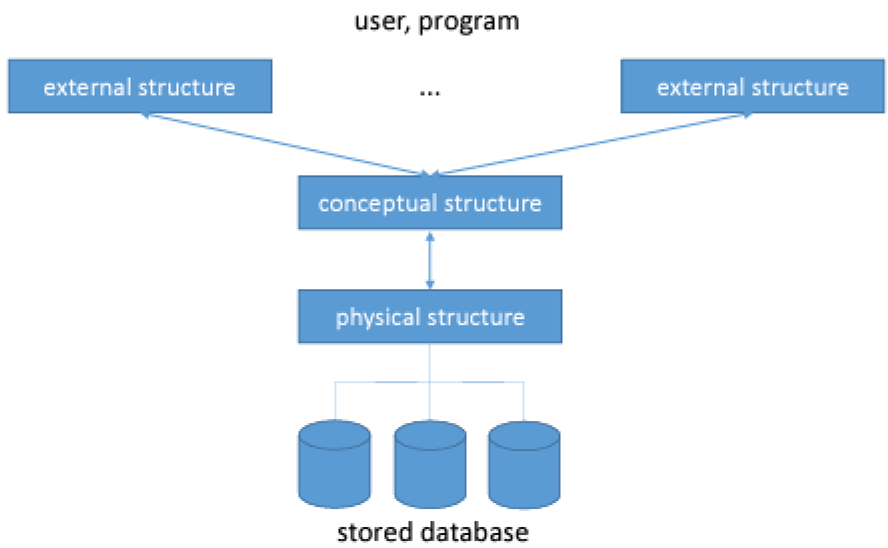
Databases Lecture Notes

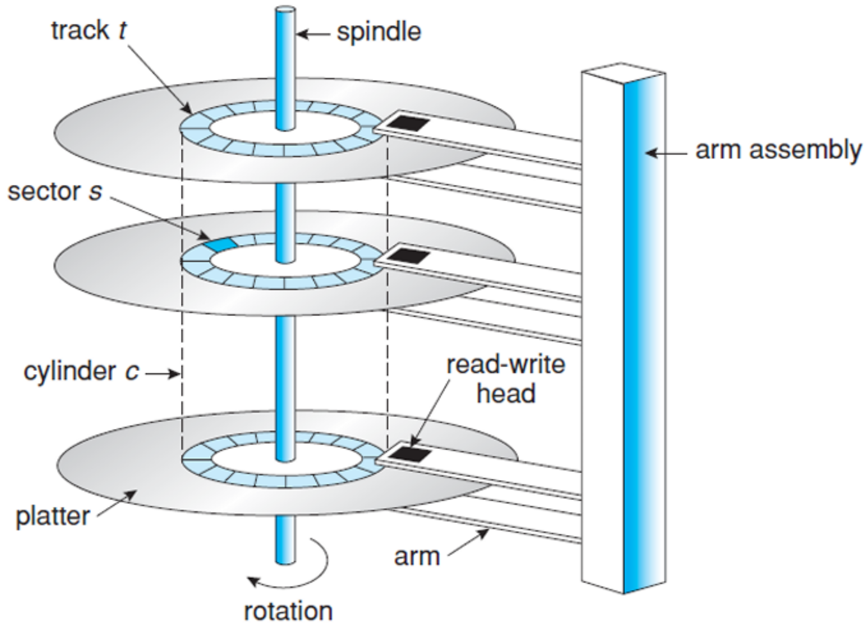
- Data Description Models
    - set of concepts and rules used to model data, such concepts describe:
        - the structure of the data
        - consistency constraints
        - relationships with other data
    - the relational model
        - the main concept: relation
            - a data collection is used according to a model, as set of relations(tables)
            - schema of a relation: relation name + for each column its name and type = table column heads
            - relation instance = table
                - rows are not ordered
                - records/tuples are distinct, but DBMS allows duplicates
                - cardinality = number of tuples
                - values of an atrribute are atomic and scalar
            - Key = a restriction defined on an entity set; a set of attributes with distinct values in the entity’s set instances
            - integrity constraints
                - conditions specified on the database schema, restricting the data that can be stored in the database
                - checked when data is changed
                - types

            - domain constraints
                - every relation instance must satisfy
            - key constraints
                - primary key
                    - a set of fields that contain the key is a superkey
                    - one primary key and multiple candidate keys
                - foreign key
        - relational database
            - collection of relations with distinct names
        - relational database schema
            - collection of schemas for the relations in the database
        - database instance
            - collection of relation instances, one/relational schema in the database
    - the entity-relationship model
        - main concepts: entities, attributes, relationships
        - entity: a piece of data(object in real world), described by attributes
        - entity set: entities with the same structure; name + list of attributes
        - attribute: name, domain of possible values, conditions to check correctness
        - key: restriction defined on an entity set; a set of attributes with distinct values in the entity set's instances
        - relationship
            - specifies an association among 2 or more entities
            - descriptive attributes can be used
        - relationship set (relationship schema)
            - describes all relationships with the same structure name , entity sets used in association, descriptive attributes

            - the schema of the the model: set of entity sets and relationship sets
        - binary relationships(between entity sets T1 and T2) - relationship types: 1:1, 1:n, m:n
            - considered as restrictions in the database, when the database is changed the system checks whether the relationship is of the specified type
- Databases and Database Management Systems
    - database contains
        - database schema
            - description of data structures used to model the data
            - kept in a database dictionary
        - a collection of data: instances of the schema
        - various components: views, procedures, functions, roles, users, etc.
    - separation between: data definition and data management
    - database design: describe an organization in terms of the data in a database
    - data analysis: answer questions about the organization by formulating queries that involve the data in the database
    - database system: database + dbms
- The Structure of a Database
    - the ANSI-SPARC arhitecture: a three-level arhitecture for a database system, proposed in 1975, in general, this model is used by the main managment systems and includes:



- the conceptual structure (the database schema)
  - information about entities and relationships among entities
- external structures: describe the data structures, used by a certain user/program; the description empoys a certain model, and the DBMS can find the data in the conceptual structure
- the physical structure (internal structure): describes the storage structures i the database(data files, indexes, etc)
  - The memory hierarchy
    - primary storage
      - cache, main memory
      - very fast access to data
      - volatile
      - currently used data
    - secondary storage
      - magnetic disks



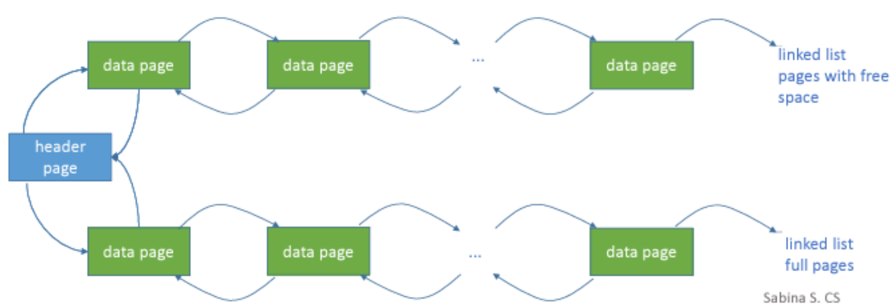
- disk block
    - sequence of contiguous bytes
    - unit for data storage
    - unt for data transfer
- tracks concentric rings containing blocks, record on one or more platters
- sectors
    - arcs on tracks
- platters
    - single-sided, double-sided
- cylinder
    - set of all tracks with the same diameter
- disk heads
    - one per recorded surface
    - head must be on top of the block to read

            - all disk heads are moved as a unit
        - time to access a desired location
            - main memory - the same for any location
            - disk - depends on where the data is stored
                - seek time + rotational delay + transfer time
                - seek time → time to move the disk head to the desired track
                - rotational delay → time for the block to get under the head
                - transfer time → time to read/write the block, once the disk head is positioned over it
    - slower storage devices
    - nonvolatile
    - disks - sequential, direct access
    - main database
  - tertiary storage
    - optical disks, tapes
    - slowest storage devices
    - nonvolatile
    - tapes
      - only sequential access
      - good for archives, backups
      - unsuitable for data that is frequently acesssed
  - disks and tapes cheaper then main memory
- managing disk space
  - disk space manager(DSM)

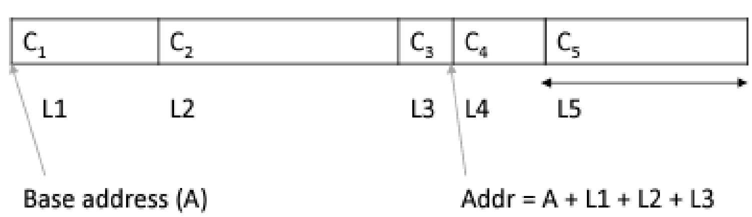
        - commands to allocate / deallocate/ read/ write a page
        - knows which pages are on which disk blocks
        - monitors disk usage, keeping track of available disk blocks
        - free blocks can be identified
            - by maintaining a linked list of free blocks
            - by mainting a bitmap with one bit/block, indicating if it is used or not
                - allows for fast identification of contigous available areas on disk
    - page
        - unit of data
        - size of a page = size of a disk block
        - R/W a page = one I/O operation
    - upper layers in the DMBS can treat the data as a collection of pages
- Buffer Manager(BM)
    - brings new data pages from disk to main memory as they are required
    - decides what main memory pages can be replaced
    - manages the available main memory
        - collection of pages called the buffer pool(BP)
        - frame
            - page in the BP
            - slot that can hold a page
            - pin_cout - nr of current users
            - dirty - bool for signaling if the page in F has been change since being brought into F

- replacement policy
  - policy that dictates the choice of replacement frames in the bp
  - LRU (least recently used), MRU, random, clock replacement, toss-immediate
- When a page is requested
  1. If the page is in the BP, increase its pin count => done
  2. Else, select a frame (FR) to replace using the replacement policies; if the old frame was dirty, write it on the disk; pin_count(FR)++, the new page is read by the BM in that frame
  3. If the BP is full, the operation may be aborted, or it waits
- It may pre-fetch pages

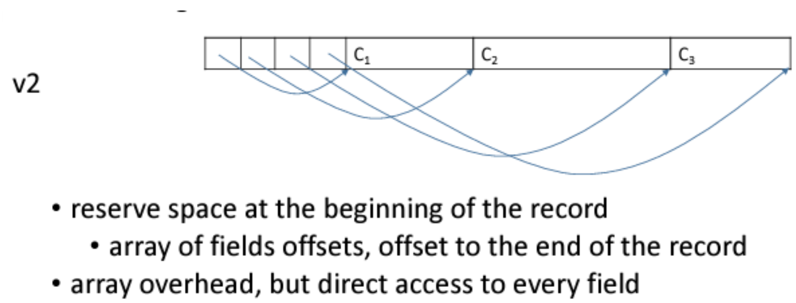
- Files of records
  - higher level layers in the DBMS treat pages as collections of record, every record has an identifier
  - Heap files
    - simplest file structure
    - not ordered
    - operations: create file, destroy file, insert record, get record by rid, delete record by rid, scan all records
    - best for record scan
    -
    - doubly linked list of pages



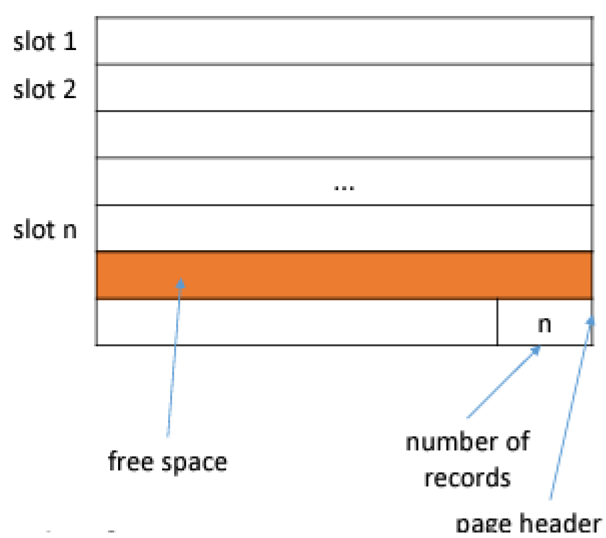
        - directory of pages
            - DBMS stores the location of the header page for each heap file
            - directory - collection of pages
            - directory entry - identifies a page in the file
            - directory entry size << page size
            - directory size << file size
            - free space managment
                - 1 bit / directory entry → page has / doesnt have free space
                - count / entry → available space on the coressponding page ⇒ efficient search of pages with enough free space when adding a variable-length record
    - sorted files
    - hashed files
    - record formats
        - fixed-length records
            - each field has a fixed length
            - fixed number of fields
            - fields stored consecutively
            - computing address - record address + length of preceding fields

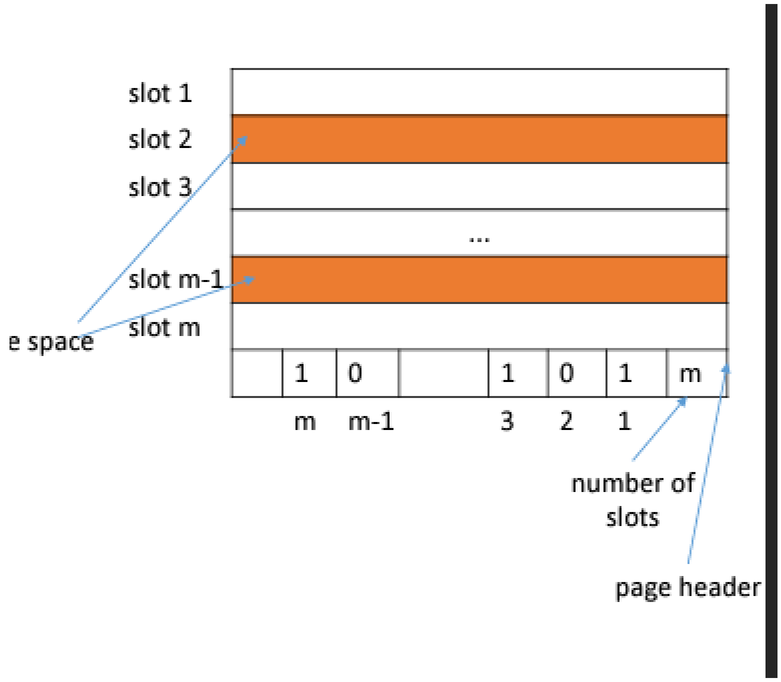


- variable-length records
  - variable-length fields
  - fileds sorted consecutevily, separated by delimiters
  - finding a field → record scan

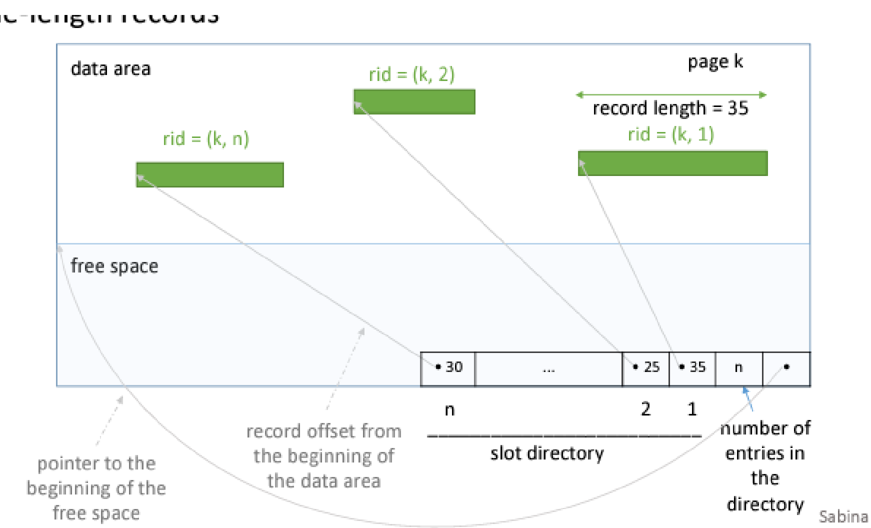


- page formats
  - fixed-length
    - record have the same size
    - uniform, consecutive slots



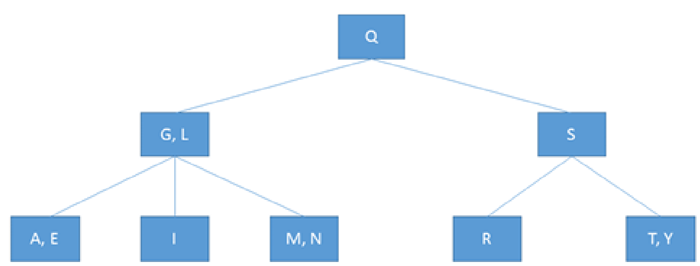


- variable-length records
  - adding a record
    - finding an empty slot of the right size
  - deleting a record
    - contiguous free space
  - a directory of slots / page
  - a pair <record offset, record length> / slot
  - moving a record on the page
    - only the record’s offset changes
    - its slot number remain unmodified

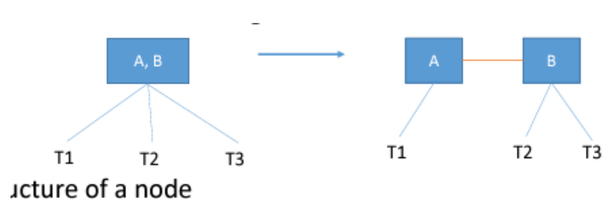


- Indexes
    - auxiliary data structure that speeds up operations which efficiently carried out given the file’s organization
    - Stored on the disk, associated with a table or view
    - search key - set of one or more attributes of the indexed file
    - and index speeds up queries with equality / range selection conditions on the search key
    - entries - record in the index <search key, rid>
    - changing data in the file ⇒ update the indexes associated with the files
    - index size - as small as possible
    - organization
        - a1: is an actual data record with search key value = k
        - a2: is a pair <k, rid>
        - a3: is a pair <k, rid_list>
    - clustered / unclustered indexes
        - clustered: the order of the data records is close / the same as the order of the data entries
        - unclustered index: index that is not clustered

    - index using a1 - clustered, using a2 are clustered only if ordered by the search key → an index that used a2 or a3 is unclusterd
    - there can be at most 1 clustered index and several unclustered indexes
- primary / secondary indexes
    - primary: includes the primary key
    - secondary: not primary
    - unique: includes a candidate key
    - primary and unique cannot contain duplicates
- Tree-structured indexing
    - 2-3 tree



        - terminal nodes - same level
        - storing a 2-3 tree
            - transform 2-3 tree into binary tree
                - nodes with 2 values are tranformed



                - structure of a node

K	ADDR	PointerL	PointerR	IND

- K - key value
- ADDR - address of the record with the current key value (address in the file)
- PointerL, PointerR - the 2 subtrees' addresses (address in the tree)

Sabina S. CS

- the memory area allocated for a node can store 2 values and 2 subtree addresses

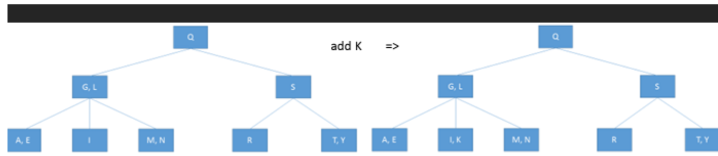
NV	K_1	$ADDR_1$	K_2	$ADDR_2$	$Pointer_1$	$Pointer_2$	$Pointer_3$

- NV – number of values in the node (1 or 2)
- K_1, K_2 – key values
- $ADDR_1$, $ADDR_2$ – the records' addresses (corresponding to K_1 and K_2)
- $Pointer_1$, $Pointer_2$, $Pointer_3$ – the 3 subtrees' addresses

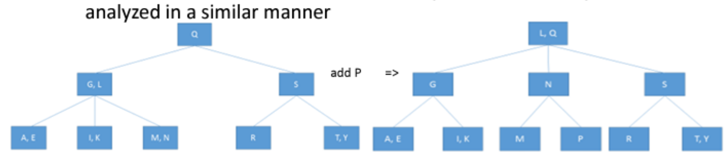
- operations
  - searching
  - tree traversal (partial, total)
  - add a new value

- if the reached terminal node has 1 value, the new value can be stored in the node

Sabina S. CS



- if the reached terminal node has 2 values, the new value is added to the node, the 3 values are sorted, the node is split into 2 nodes: one node will contain the smallest value, the 2nd node - the largest value, and the middle value is attached to the parent node; the parent is then analyzed in a similar manner



Sabina S. CS

- delete a value

1. search for K_0; if K_0 appears in an inner node, change it with a neighbor value K_1 from a terminal node (there is no other value between K_0 and K_1)

- K_1's previous position (in the terminal node) is eliminated

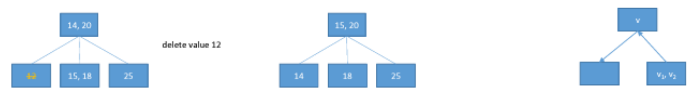
- e.g., remove 8:

2. perform this step until case a / b occurs

a. if the current node (from which a value is removed) is the root or a node with 1 remaining value, the value is eliminated; the algorithm ends

Sabina S. CS

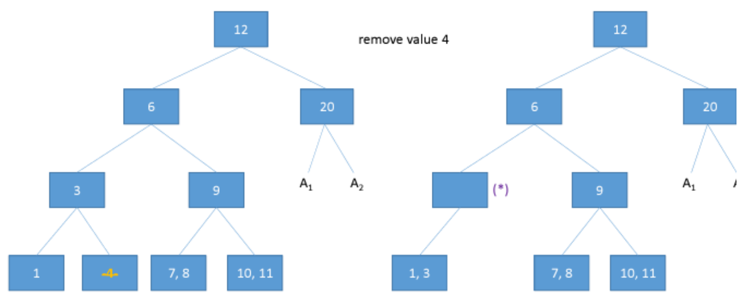
b. if the delete operation empties the current node, but 2 values exist in one of the sibling nodes (left / right), 1 of the sibling's values is transferred to the parent, 1 of the parent's values is transferred to the current node; the algorithm ends



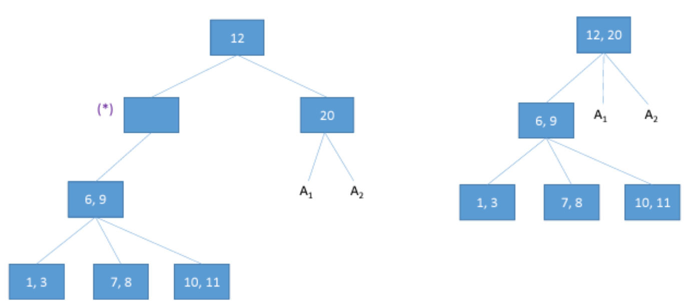
c. if the previous cases do not occur (current node has no values, sibling nodes have 1 value each), then the current node is merged with a sibling and a value from the parent node; case 2 is then analyzed for the parent

- if the root is reached and it has no values, it is eliminated and the current node becomes the root

- example: case c for the node marked with (*)



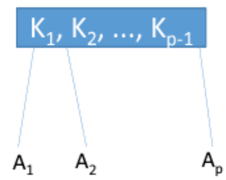
Sabina S. CS



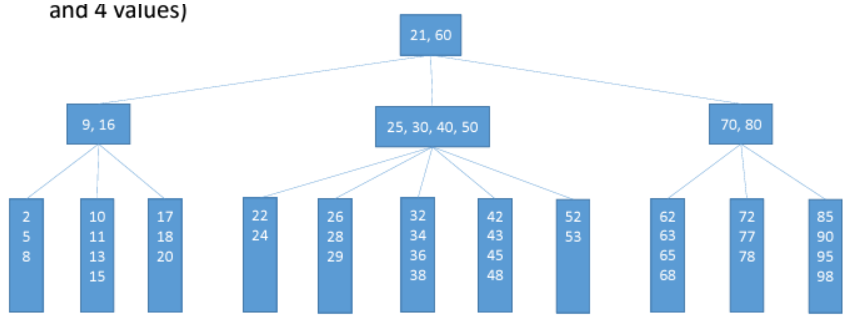
- b trees

B-tree of order m

1. if the root is not a terminal, it has at least 2 subtrees
2. all terminal nodes – same level
3. every non-terminal node – at most m subtrees
4. a node with p subtrees has p-1 ordered values (ascending order): $K_1<K_2<\ldots<K_{p-1}$
   - A_1: values less than K_1
   - A_i: values between K_{i-1} and K_i, i=2,...,p-1
   - A_p: values greater than K_{p-1}
5. every non-terminal node – at least $\left\lceil \frac{m}{2} \right\rceil$ subtrees



and 4 values)



- storing
  - transformed into a binary tree
  - the memory area allocated for a node can store the maximum number of values and subtree addresses

NV	K_1	$ADDR_1$	...	K_{m-1}	$ADDR_{m-1}$	$Pointer_1$	...	$Pointer_m$

- NV - number of values in the node
- K_1, ..., K_{m-1} - key values
- $ADDR_1$, ..., $ADDR_{m-1}$ - the records' addresses (corresponding to the key's values)
- $Pointer_1$, ..., $Pointer_m$ – subtree addresses

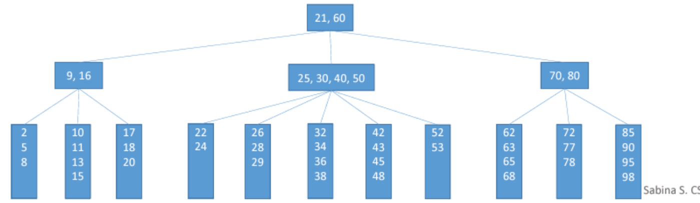
Sabina S. CS

- operations
  - searching for a value
  - adding a value

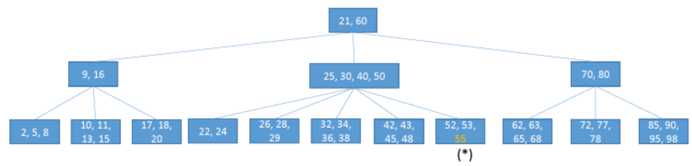
- adding a new value

  1. values in the tree must be distinct (the new value should not exist in the tree); perform a test (search for the value in the tree)
  - if the new value can be added, the search ends in a terminal node

  2. if the reached terminal node has less than m-1 values, the new value can be stored in the node, e.g., 55 is added to the tree below:

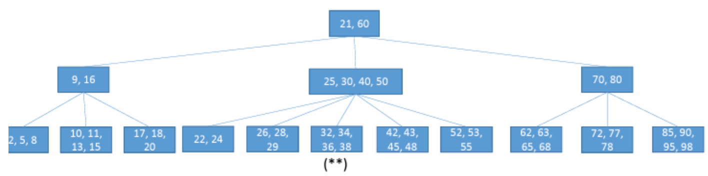


the resulting tree is shown below:



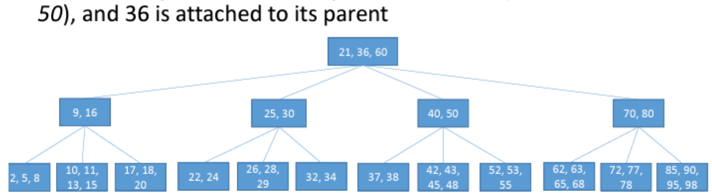
- 55 belongs to the node marked with (*), which can store at most 4 values

3. if the terminal node already has m-1 values, the new value is attached to the node, the m values are sorted, the node is split into 2 nodes, and the middle value (median) is attached to the parent node; the parent is then analyzed in a similar manner
   - e.g., add 37 to the tree below

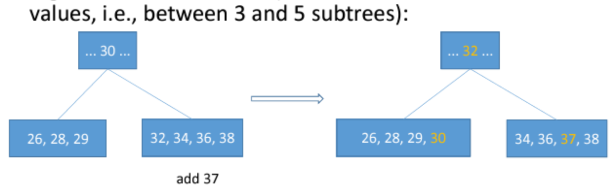


   - the node marked with (**) should contain values *32, 34, 36, 37, 38*

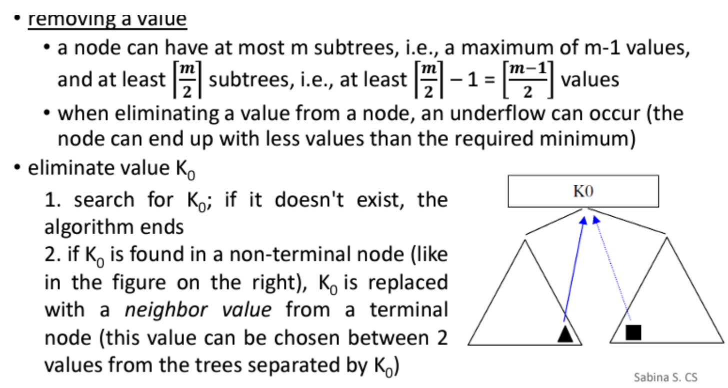
- since the node's capacity is exceeded, it is split into nodes *32, 34*, and *37, 38*, and 36 is attached to the parent node (with values *25, 30, 40, 50*)
- in turn, the parent must be split into 2 nodes (values *25, 30*, and *40, 50*), and 36 is attached to its parent

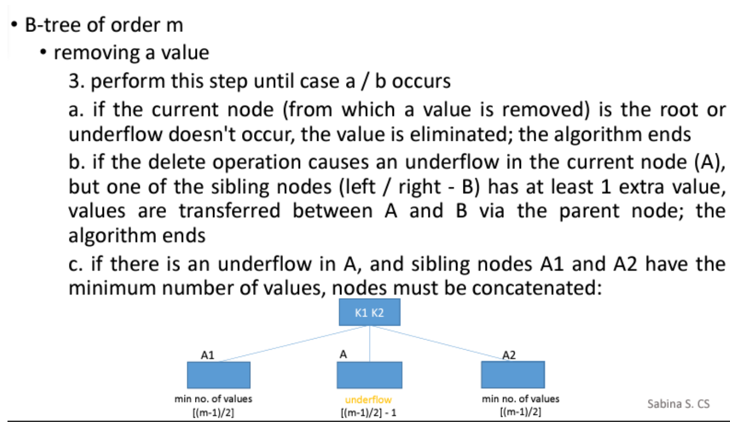


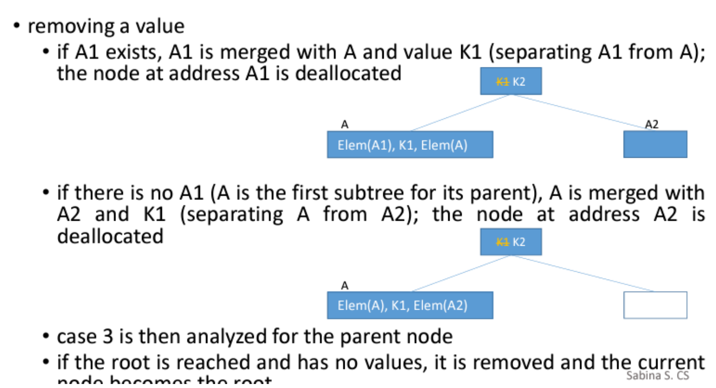
- <u>optimizations</u>
  - before performing a split - analyze whether one or more values can be transferred from the current node (with m-1 values) to a sibling node
  - e.g., B-tree of order 5 (non-terminal node - between 2 and 4 values, i.e., between 3 and 5 subtrees):



- removing a value







- tree traversal

- storing blocks

- obs. a block stores a node from a B-tree
- e.g.:
    - key size: 10b
    - record address / node address: 10b
    - NV value (number of values in the node): 2b
    - block size: 1024b (10b for the header)
- then: 2+(m-1)*(10+10)+m*10=1024-10 => m=34
- if the size of a block is 2048b and the other values are unchanged, then the order of the tree is m = 68, i.e., a node can have between 33 and 67 values

- B-tree of order m
- the maximum number of required blocks (from the file that stores the B-tree) when searching for a value - the maximum number of levels in the tree; for m=68, if the number of values is 1.000.000, then:
    - the root node (on level 0) contains at least 1 value (2 subtrees)
    - on the next level (level 1) - at least 2 nodes * 33 values/node = 66 values
    - level 2 – at least 2*34 nodes * 33 values/node = 2.244 values
    - level 3 – at least 2*34*34 nodes * 33 values/node = 76.296 values
    - level 4 – at least 2*34*34*34 nodes * 33 values/node = 2.594.064 values, which is greater than the number of existing values => this level does not appear in the tree

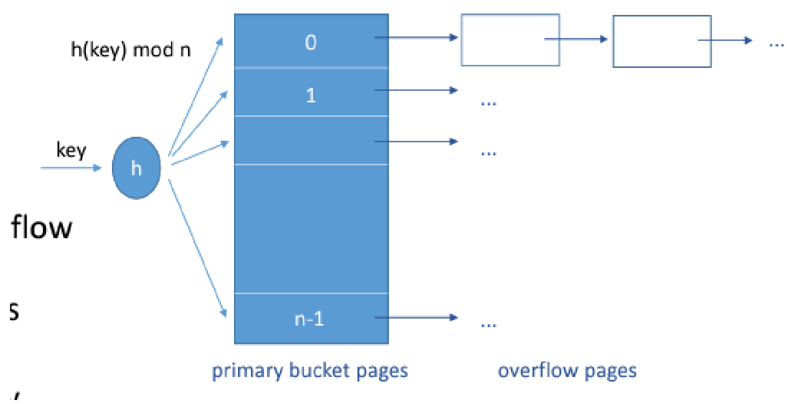
=> at most 4 levels in the tree

- after at most 4 block reads and a number of comparisons in main memory, it can be determined whether the value exists (the corresponding record's address can then be retrieved) or the search was unsuccessful

- b+ tree
    - b tree variant
    - last level contains all values (key values and the record's addresses)
    - some key values can also appear in non-terminal nodes, without the records addresses, their purpose is to separate values from terminal nodes
    - terminal nodes are maintained in a doubly linked list (data can be easily scanned)
    - in practice
        - concept of order → physical space criterion
        - variable length search key → variable length entries → variable length of entries/ page
        - prefix key compression

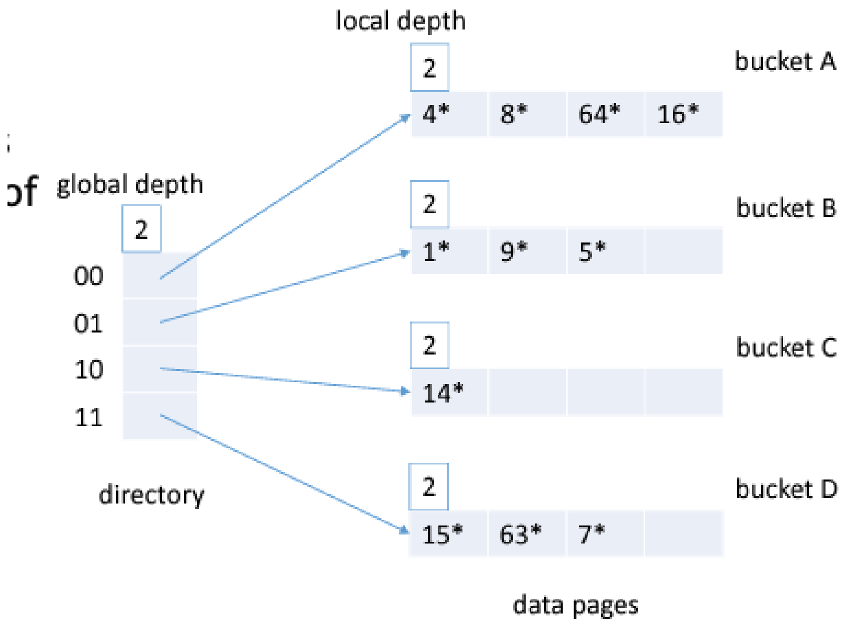
- hashed-based indexing

- hashing functions: maps search key values into a range of bucket numbers
- hashed file
    - search key (field(s) of the file)
    - records grouped into buckets
    - determine record r's bucket
        - apply hash function to search key
    - quick location of records with given search key value
- ideal for equality selections
- static hashing
    - buckets 0 to n-1
    - bucket



        - one primary page
        - possibly extra overflow plages
        - data entries in buckets a1/a2/a3
    - search for a data entry
        - apply hashing function to identify the bucket
        - search the bucket
        - possible optimization

            - entries sorted by search key
        - add data entry
            - apply hashing function to identify bucket
            - add the entry to the bucket
            - if there is no space in the bucket
                - allocate an overflow page
                - add data entry to the page
                - add the overflow page to the bucket’s overflow chain
        - delete a data entry
            - apply hashing function to identify the bucket
            - search the bucket to locate the data entry
            - remove the entry from the bucket
            - if the data entry is the last one on its overflow page
                - remove the overflow page from its overflow chain
                - add the page to a free pages list
        - good hashing function: key values are uniformly distributes over the set of buckets
    - extendible hashing



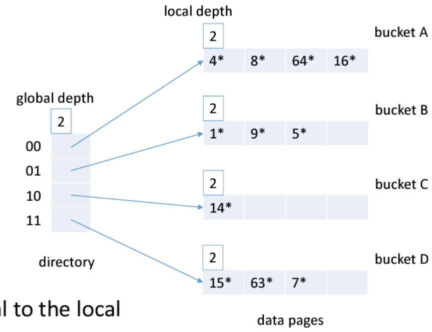
- dynamic hashing technique
- directory of pointers to buckets
- double the size of the number of buckets
  - double the directory
  - split overflowing bucket
- directory: array of 4 elents
- directory elem: pointer to bucket

  - entry r with key value K
  - $h(K) = (\ldots a_2a_1a_0)_2$
  - $nr = a_1a_0$, i.e., last 2 bits in $(\ldots a_2a_1a_0)_2$, nr between 0 and 3
  - directory[nr]: pointer to desired bucket

- global depth

* extendible hashing

- global depth *gd* of hashed file
  - number of bits at the end of hashed value interpreted as an offset into the directory
  - kept in the header
  - depends on the size of the directory
    - 4 buckets => gd = 2
    - 8 buckets => gd = 3
- initially, the global depth is equal to the local depth of every bucket



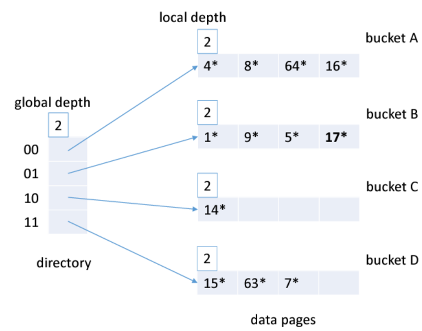
- adding an element

* extendible hashing

- insert entry
  - find bucket

  a. bucket has free space => the new value can be added
  - example: add data entry with hash value 17 to bucket B

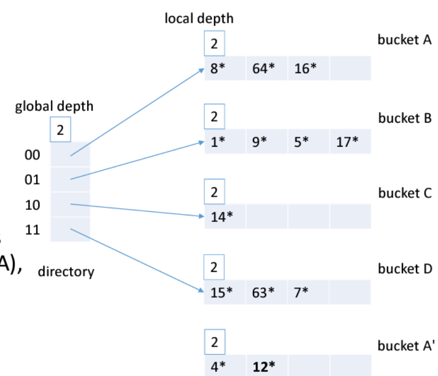
obs. data entry with hash value 17 is denoted as 17*



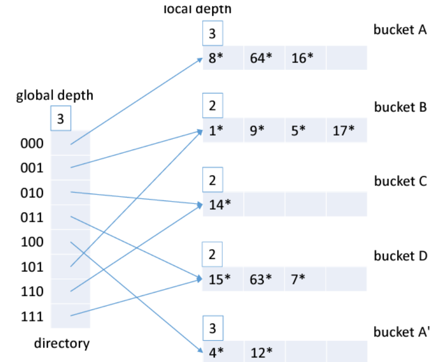
* extendible hashing

- insert entry

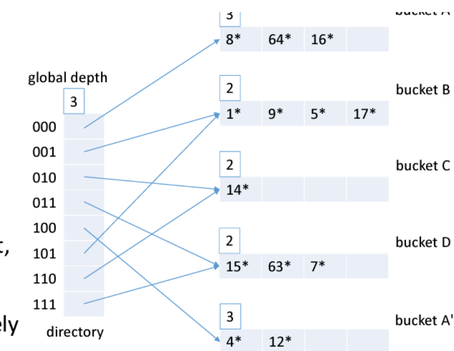
  b. bucket is full
  - example: add entry 12*, bucket A full
  - split bucket A
    - allocate new bucket A'
    - redistribute entries across A & A' (the split image of A), by taking into account the last 3 bits of h(K)



* extendible hashing
- insert entry
  - b. bucket is full
  - if gd = local depth of bucket being split => double the directory, gd++
  - 3 bits are needed to discriminate between A & A', but the directory has only enough space to store numbers that can be represented on 2 bits, so it is doubled
  - increment local depth of bucket: LD(A) = 3
  - assign new local depth to bucket's split image: LD(A') = 3

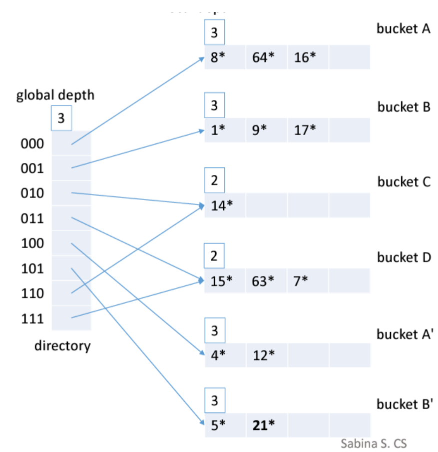


- insert entry
  - b. bucket is full
  - *corresponding elements*
    - 000, 100
    - 001, 101
    - 010, 110
    - 011, 111
    - point to the same bucket, except for 000 and 100, which point to A and split image A', respectively



* extendible hashing
- insert entry
  - b. bucket is full
  - example: add 21*
  - it belongs to bucket B, which is already full, but its local depth is 2 and gd = 3

  => split B, redistribute entries, increase local depth for B and its split image; directory isn't doubled, gd doesn't change



○ removing an element

- delete entry
  - locate & remove entry
  - if bucket is empty:
    - merge bucket with its split image, decrement local depth
  - if every directory element points to the same bucket as its split image:
    - halve the directory
    - decrement global depth

- obs

  - obs 1. 2^{gd-ld} elements point to a bucket Bk with local depth ld
    - if gd=ld and bucket Bk is split => double directory
  - obs 2. manage collisions - overflow pages

  - bucket split accompanied by directory doubling
    - allocate new bucket page nBk
    - write nBk and bucket being split
    - double directory array (which should be much smaller than file, since it has 1 page-id / element)
      - if using *least significant bits* (last gd bits) => efficient operation:
        - copy directory over
        - adjust split buckets' elements

  Sabina S. CS

  * extendible hashing
  - equality selection
  - if directory fits in memory:
    => 1 I/O (as for Static Hashing with no overflow chains)
  - otherwise
    - 2 I/Os

  - example: 100 MB file, entry = 50 bytes => 2.000.000 entries
  - page size = 8 KB => approx. 160 entries / bucket

  => need 2.000.000 / 160 = 12.500 directory elements

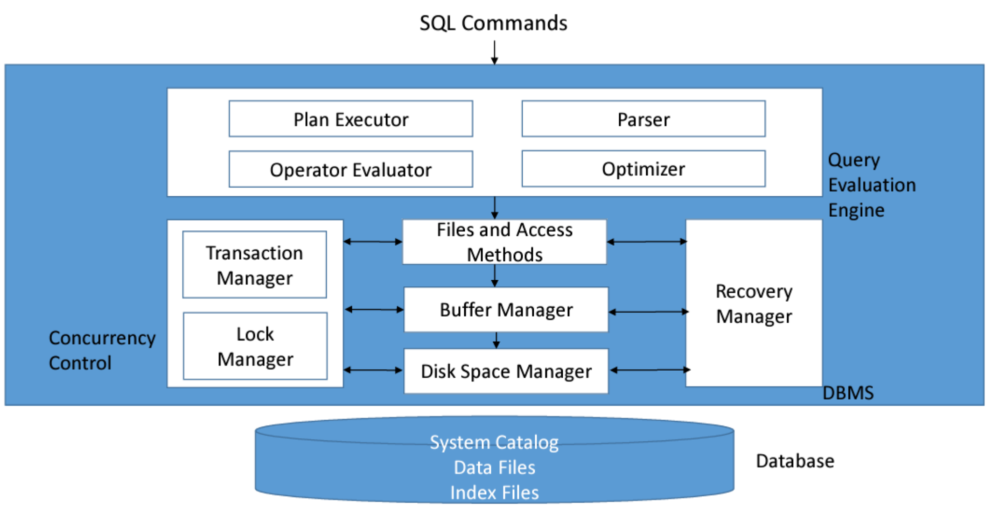
- Logical Independence and Physical Independence
  - data independence: 3 levels of abstraction —> applications are insulated from changes in the data structure/storage
  - logical data independent: programs using data from the database are not affected by changes in the conceptual structure
  - physical data independence - the capacity to change the internal schema without having to change the conceptual schema
- Functions of DBMS
  - database definition: DDL
  - data managment: insert, update, delete, quering
  - data administration:
    - access authorization,
    - usage monitoring,

- performance monitoring and optimization

- the protection of the database:
  - confidentiality(protection against unauthrorized data access)
  - integrity(protection against inconsistent changes)

- Types of Database Users
  - administrators
  - designers
  - application users
  - application programmers

- The Arhitecture of a DBMS



  - optimizer: produces an efficient execution plan for query evaluation, taking into account storage information
  - file & access methods, buffer/disk manager: abstractizaation of files, bringing pages from the disk into memory, managing disk space
  - Transaction Manager, Lock Manager: concurrency control, monitoriting lock requests, granting lock when database obj become available
  - recovery manager: recovery after a crash

- Managing Relational Dabases with SQL

- defining components:
  - create
    - create table - defines a new table
      - restrictions associated with a column/table*
        - not null
        - primary key*
        - unique*
        - check(condition)*
        - foreign key*
          - no action
          - set null
          - set default
          - cascade
  - alter
    - alter table - changes the structure of a defined table
      - add/change/remove a column
      - add/remove a constraint
  - drop
    - drop table - destroys a table
- managing and retrieving data
  - select
  - insert

```
insert into table_name [(column_list)] values (value_li
insert into table_name [(column_list)] subquery
```

  - update

```
update table_name
set column_name=expression
[where condition]
```

- delete

```
delete from table_name
[where condition]
```

- filtering conditions

```
--elementary condition
expresssion comparasion_operator expression
expression [not] between valmin and valmax
expression [not] like pattern --("%" - any substring, '
expression is [not] null
expression [not] in (value,..) or (subquery)
expression comparasion_operator {all | any} {subquery}
[not] exists {subquery}
not condition
condition1 and condition2
condition2 or condition2
```

- 3-Values Logic

3-Valued Logic

- truth values: *true*, *false*, *unknown*

	TRUE	FALSE	NULL
NOT	FALSE	TRUE	NULL

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

- managing transactions
  - start transaction
  - commit
  - rollback

- Quering Relational Databases using SQL
  - basic SQL query

```
select [distinct] select_list
from from_list
where qualification -- optional
```

  - conceptual evaluation strategy
    - compute cross product of tables in the from_list
    - remove the rows that don’t meet qualification
    - eliminate unwanted columns, those that dont appear in select_list
    - if distinct —> remove duplicates
  - logical processing
    - from → where → group by → having → select → distinct → order by → top
  - expresssions in select

- range variable
  - alias used for a table in a sql query
  - needed when a relation a appears more then once in the from clause
- as and = can be used to name fields in the result set
- SELECT can contain arithmetic operations
- SELECT, FROM - mandatory;
- nested queries in the where clause
  - in - it test whether a value belongs to a set of elements, the latter can be explicitly specified or generated by a query
  - exists = it tests wheter a set is non-empty
  - any ↔ in, true if condition is true for at least one item in the subquery result
  - all , !all ↔ not in, true if condition is true for all items in the subquery
- union, intersection, set-difference
- joins

```
--inner join
source1 [alias] [inner] join source2 [alias] on condition
--left outer join
source1 [alias] left [outer] join source2 [alias] on condi
--right outer join
source1 [alias] right [outer] join source2 [alias] on cond
--full outer join
source1 [alias] full [outer] join source2 [alias] on condi
--other joins
source1 [alias] join source2 [alias] using (column_list)
source1 [alias] natural join source2
source1 [alias] cross join source2 [alias]
```

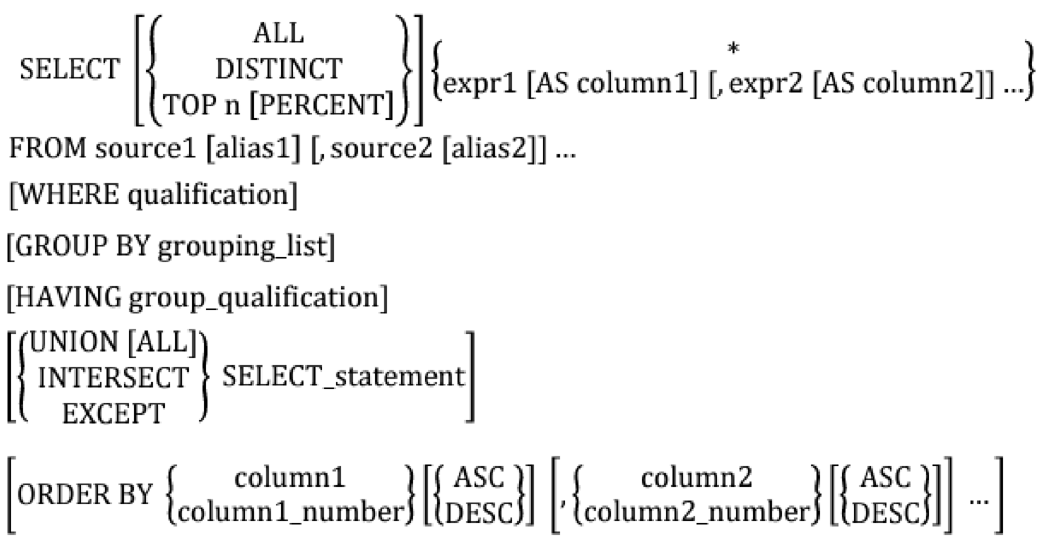
- subquery in the from clause

  If we have a FROM query, we need to give it an alias

- group by, having
  - HAVING cannot contain row-level conditions because it works with groups
  - Can’t group by a subquery

  ```
  select [disctinct] select_list
  from from_list
  where qualification
  group by grouping_list
  [having group_qualification]
  ```

  - every row in the result correspounds to a group
  - select_list columns must appear in grouping_list
- aggregation operators: count, avg, sum, min, max
- order by, top [percent]



- stored procedures
  - contains a group of Transact-SQL statement

  ```
  create procedure <SPName> as
   --sequence of sql statements
  go
  ```

```
[exec] <SPName>
```

    - can have parameters, keyword output for output parameter
    - raiseerror: generate error message
  - gloval variables
    - @@ error - the erro rnumber for the last executed T-SQL statement, 0 no error
    - identity - the last inseted identity value
    - rowcount - the number of rows affected by the last statement
    - server name - the name of the local server on which sql server is running
    - spid - session id for current user
    - version - system and build information
  - output clause - provide access to added/modified/deleted records
    - inserted, deleted table - temporary tables
  - cursors
    - used when processing row-by-row, used in Transact-SQL scripts, stored procedures, triggers

- extended Transact-SQL syntax:

```
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]
     [ FORWARD_ONLY | SCROLL ]
     [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
     [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
     [ TYPE_WARNING ]
     FOR select_statement
     [ FOR UPDATE [ OF column_name [ ,...n ] ] ]
[;]
```

- FETCH
    - fetches a certain row from a cursor;
    - once the cursor is positioned on a row, various operations can be performed on the row;
    - FETCH options to obtain certain rows:
        - FETCH FIRST – returns the first row from the cursor;
        - FETCH NEXT – the row immediately following the current row;
        - FETCH PRIOR – the row before the current row;
        - FETCH LAST – the last row in the cursor;
        - FETCH ABSOLUTE n, n integer:
            - $n > 0$: the n^{th} row starting with the first row in the cursor;
            - $n < 0$: the n^{th} row before the last row in the cursor;
            - $n = 0$: no rows;
        - FETCH RELATIVE n, n integer:
            - $n > 0$: the row that is *n* rows after the current row;
            - $n < 0$: the row that is *n* rows before the current row;
            - $n = 0$: the current row.

- user-defined functions
    - scalar - return scalar value

example:

```
CREATE FUNCTION ufNoStudents(@age INT)
RETURNS INT AS
BEGIN
  DECLARE @no INT
  SET @no = 0
  SELECT @no= COUNT(*)
  FROM Students
  WHERE age = @age
  RETURN @no
END
GO

PRINT dbo.ufNoStudents(20)
```

b. inline table-valued functions

- inline table-valued
    - return a table
    - can be called in the from clause
    - on statement

```
CREATE FUNCTION ufStudentsNames(@age INT)
RETURNS TABLE
AS
  RETURN
    SELECT sname
    FROM Students
    WHERE age = @age
GO

SELECT *
FROM ufStudentsNames(20)
```

    - multi-statement table-valued functions
        - returns a table
        - more then 1 statement

```
CREATE FUNCTION ufCoursesFilteredByCredits(@credits INT)
RETURNS @CoursesCredits TABLE (cid INT, cname VARCHAR(70))
AS
BEGIN
  INSERT INTO @CoursesCredits
  SELECT cid, cname
  FROM Courses
  WHERE credits = @credits

  IF @@ROWCOUNT = 0
    INSERT INTO @CoursesCredits
    VALUES (0,'No courses found with specified number of credits.')

  RETURN
END
GO

SELECT *
FROM ufCoursesFilteredByCredits(5)
```

  - views
    - create a virtual table representing data from one for more tables in an alternative manner
    - most 1024 columns

- syntax:

```
CREATE VIEW view_name
AS SELECT_statement
```

- example:

```
CREATE OR ALTER VIEW vExaminations
AS
SELECT S.sid, S.sname, S.sgroup, C.cid, C.cname
FROM Students S INNER JOIN Exams E ON S.sid= E.studentid
  INNER JOIN Courses C ON E.courseid = C.cid
GO

SELECT *
FROM vExaminations
```

- triggers
  - special type of stored procedure
  - automatically exectuted in response to DML or DDL
  - they are not executed directly

```
CREATE TRIGGER <trigger_name>
 ON { table | view}
 [ WITH <dml_trigger_option> [ ,...n ] ]
 { FOR | AFTER | INSTEAD OF }
 { [INSERT] [,] [UPDATE] [,] [DELETE] }
 [ WITH APPEND ]
 [ NOT FOR REPLICATION ]
AS
 { sql_statement [;] [ ,...n ] |
EXTERNAL NAME <method specifier[;] > }
```

- the moment a trigger is executed is specified through one of the options:
  - FOR, AFTER - the DML trigger is fired only when all the operations specified in the triggering statement have launched successfully (multiple such triggers can be defined);
  - INSTEAD OF - the DML trigger is executed instead of the triggering statement;

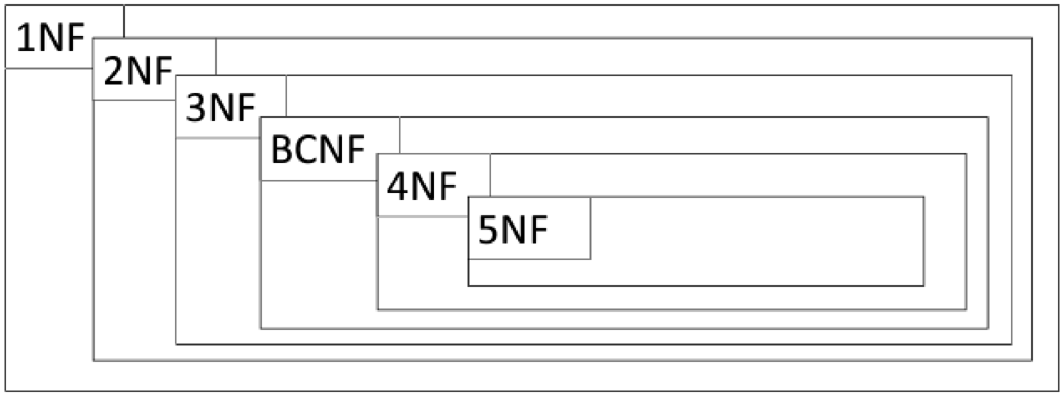
```
CREATE TRIGGER When_adding_prod
  ON Products
  FOR INSERT
AS
BEGIN
  INSERT INTO BuyLog(PName, OperationDate, Quantity)
  SELECT PName, GETDATE(), Quantity
  FROM inserted
END
GO
```

```
CREATE TRIGGER [dbo].[When_deleting_prod]
  ON [dbo].[Products]
  FOR DELETE
```



```
AS
BEGIN
  SET NOCOUNT ON;
  INSERT INTO SellLog(PName, OperationDate, Quantity)
  SELECT PName, GETDATE(), Quantity
  FROM deleted
END
GO
```

- system catalog
    - stores data about objects in the database
    - managed by the server
- Functional Dependencies. Normal Forms
    - most common normal forms



Sabina S. CS

- 1NF
  - if and only if it doesnt have any repeating attributes
  - decomposition
    - let $R[A]$ be a relation; A - the set of attributes
    - let α be a repeating attribute in R (simple or composite)
    - R can be decomposed into 2 relations, such that α is not a repeating attribute anymore
    - if K is a key in R, the two relations into which R is decomposed are:

$$R'[K \cup \alpha] = \Pi_{K \cup \alpha}(R)$$
$$R''[A - \alpha] = \Pi_{A-\alpha}(R)$$

- 2NF
  - is in 1NF
  - every non-prime attribute is fully functionally dependent on every key of the relation
  - decomposition
    - relation $R[A]$ (A - the set of attributes), K - a key
    - β non-prime, β functionally dependent on α, $\alpha \subset K$ (β is functionally dependent on a proper subset of attributes from a key)
    - the $\alpha \rightarrow \beta$ dependency can be eliminated if R is decomposed into the following 2 relations:

$$R'[\alpha \cup \beta] = \Pi_{\alpha \cup \beta}(R)$$
$$R''[A - \beta] = \Pi_{A-\beta}(R)$$

- 3NF
  - is in 2NF
  - no non-prime attribute is transitively dependent on any key in the relation
  - or: iff, for every non-trivial functional dependency X → A that holds over R: X is a superkey, or A is a prime attribute
- BCNF
  - iff every determinant (for a functional dependency) is a key
  - informal definition: determinants are not too big; only non-trivial functional dependencies are considered
- 4NF
  - is in BCNF

Definition. A relation R is in 4NF iff, for every multi-valued dependency $\alpha \rightrightarrows \beta$ that holds over R, one of the statements below is true:
- $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$, or
- α is a superkey.

- if $R[\alpha, \beta, \gamma]$ and $\alpha \rightrightarrows \beta$ (non-trivial, α not a superkey), R is decomposed into the following relations:

$R_1[\alpha, \beta] = \Pi_{\alpha \cup \beta}(R)$
$R_2[\alpha,\gamma] = \Pi_{\alpha \cup \gamma}(R)$

- 5NF

Definition. Relation R is in 5NF iff every non-trivial JD is implied by the candidate keys in R.
- JD $* \{\alpha_1, \alpha_2, \ldots, \alpha_{\mathrm{m}}\}$ on R is trivial iff at least one α_i is the set of all attributes of R.
- JD $* \{\alpha_1, \alpha_2, \ldots, \alpha_{\mathrm{m}}\}$ on R is implied by the candidate keys of R iff each α_i is a superkey in R.

- Functional Dependency

Definition. Let $R[A_1, A_2, \ldots, A_n]$ be a relation and α, β two subsets of attributes of R. The (simple or composite) attribute α functionally determines attribute β (simple or composite), notation:

$$\alpha \rightarrow \beta$$

if and only if every value of α in R is associated with a precise, unique value for β (this association holds throughout the entire existence of relation R); if an α value appears in multiple rows, each of these rows will contain the same value for β:

$$\Pi_\alpha(r) = \Pi_\alpha(r') \ \ implies \ \ \Pi_\beta(r) = \Pi_\beta(r')$$

- in the dependency $\alpha \rightarrow \beta$, α is the *determinant*, and β is the *dependent*
- the functional dependency can be regarded as a property (restriction) that must be satisfied by the database throughout its existence: values can be added / changed in the relation only if the functional dependency is satisfied

- **Just by looking at an instance, we can only say that a func. dep is satisfied or not, we can't conclude that it's specified on the schema**
- problems when having functional dependencies
  - wasting space
  - insert, update, delete anomalies
- properties:
  - reflexivity
  - transitivity
- A is prime attribute if there is a key K and A included in K(K can be comoposite key, A can itself be a key)
- Fully Functional dependency

Definition. Let $R[A_1, A_2, \ldots, A_n]$ be a relation, and let α, β be two subsets of attributes of R. Attribute β is *fully functionally dependent on* α if:

- β is functionally dependent on α (i.e., $\alpha \rightarrow \beta$) and
- β is not functionally dependent on any proper subset of α, i.e., $\forall\, \gamma \subset \alpha$, $\gamma \rightarrow \beta$ does not hold.

- transitive dependency

Definition. An attribute Z is transitively dependent on an attribute X if $\exists Y$ such that $X \to Y$, $Y \to Z$, $Y \to X$ does not hold (and Z is not in X or Y).

- compute the closure of F: F+
  - the set F+ contains all the functional dependencies implied by F
  - F implies a functional dependency f if f holds on every relation that satisfies f
  - compute F+ with Armstrong's Axioms

- α, β, γ - subsets of attributes of A

1. reflexivity: if $\beta \subseteq \alpha$, then $\alpha \to \beta$
2. augmentation: if $\alpha \to \beta$, then $\alpha\gamma \to \beta\gamma$
3. transitivity: if $\alpha \to \beta$ and $\beta \to \gamma$, then $\alpha \to \gamma$

- rules derived from Armstrong's axioms

4. union: if $\alpha \to \beta$ and $\alpha \to \gamma$, then $\alpha \to \beta\gamma$

$$\left.\begin{array}{l} \alpha \to \beta \underset{\text{augmentation}}{\Rightarrow} \alpha\alpha \to \alpha\beta \\ \\ \alpha \to \gamma \underset{\text{augmentation}}{\Rightarrow} \alpha\beta \to \beta\gamma \end{array}\right\} \underset{\text{transitivity}}{\Rightarrow} \quad \alpha \to \beta\gamma$$

5. decomposition: if $\alpha \to \beta\gamma$, then $\alpha \to \beta$ and $\alpha \to \gamma$

$$\left.\begin{array}{l} \alpha \to \beta\gamma \\ \beta\gamma \to \beta \text{ (reflexivity)} \end{array}\right\} \underset{\text{transitivity}}{\Rightarrow} \quad \alpha \to \beta \text{ (}\alpha \to \gamma \text{ can similarly be shown to hold)}$$

6. pseudotransitivity: if $\alpha \rightarrow \beta$ and $\beta\gamma \rightarrow \delta$, then $\alpha\gamma \rightarrow \delta$

$$\left.\begin{array}{l}\alpha \rightarrow \beta \Rightarrow \alpha\gamma \rightarrow \beta\gamma \\ \beta\gamma \rightarrow \delta\end{array}\right\} \underset{\text{transitivity}}{\Rightarrow} \quad \alpha\gamma \rightarrow \delta$$

• $\alpha, \beta, \gamma, \delta$ - subsets of attributes of A

- compute the closure of a set of attributes under a set of functional dependencies

```
closure := α;
repeat until there is no change:
  for every functional dependency β → γ in F
    if β ⊆ closure
      then closure := closure ∪ γ;
```

- compute the minimal cover for a set of dependencies

Definition: F, G - two sets of functional dependencies; F and G are equivalent (notation $F \equiv G$) if $F^+ = G^+$.

Definition: F - set of functional dependencies; a minimal cover for F is a set of functional dependencies F_M that satisfies the following conditions:

1. $F_M \equiv F$
2. the right side of every dependency in F_M has a single attribute;
3. the left side of every dependency in F_M is irreducible (i.e., no attribute can be removed from the determinant of a dependency in F_M without changing F_M's closure);
4. no dependency f in F_M is redundant (no dependency can be discarded without changing F_M's closure).

- multi-valued dependency

Definition. Let $R[A]$ be a relation with the set of attributes $A = \alpha \cup \beta \cup \gamma$. The multi-valued dependency $\alpha \rightrightarrows \beta$ (read α *multi-determines* β) is said to hold over R iff each value u of α is associated with a set of values v for β: $\beta(u) = \{v_1, v_2, \dots, v_n\}$, and this association holds regardless of the values of γ.

Property. Let $R[A]$ be a relation, $A = \alpha \cup \beta \cup \gamma$. If $\alpha \rightrightarrows \beta$, then $\alpha \rightrightarrows \gamma$.

- join dependency

Definition. Let $R[A]$ be a relation and $R_i[\alpha_i]$, i=1,2, ...,m, the projections of R on α_i. R satisfies the join dependency $* \{\alpha_1, \alpha_2, \dots, \alpha_m\}$ iff $R = R_1 * R_2 * \cdots * R_m$.

- Relational Algebra
  - conditions
    - attribute_name relational_operator value
    - attribute_name is [not] in single_column_relation
    - relation {is [not] in | = | <>} relation
    - (condition),
    - not condition, cond1 and/or cond2
  - operators
    - selection

- notation: $\sigma_C(R)$
- resulting relation:
  - schema: R's schema
  - tuples: records in R that satisfy condition C
- equivalent SELECT statement

```
SELECT *
FROM R
WHERE C
```

- projection

  - notation: $\pi_\alpha(R)$
  - resulting relation:
    - schema: attributes in α
    - tuples: every record in R is projected on α
  - α can be extended to a set of expressions, specifying the columns of the relation being computed
  - equivalent SELECT statement

```
SELECT DISTINCT α
FROM R

SELECT α
FROM R          -- algebra on bags
```

- cross-product

  - notation: $R_1 \times R_2$
  - resulting relation:
    - schema: the attributes of R_1 followed by the attributes of R_2
    - tuples: every tuple r_1 in R_1 is concatenated with every tuple r_2 in R_2
  - equivalent SELECT statement

```
SELECT *
FROM R1 CROSS JOIN R2
```

- union, set-difference, intersection

- notation: $R_1 \cup R_2$, $R_1 - R_2$, $R_1 \cap R_2$
- R_1 and R_2 must be union-compatible:
  - same number of columns
  - corresponding columns, taken in order from left to right, have the same domains
- equivalent SELECT statements

```
SELECT *        SELECT *        SELECT *
FROM R1         FROM R1         FROM R1
UNION           EXCEPT          INTERSECT
SELECT *        SELECT *        SELECT *
FROM R2         FROM R2         FROM R2
```

- join operators

  - *condition join* (or *theta join*)
    - notation: $R_1 \otimes_\Theta R_2$
    - result: the records in the cross-product of R_1 and R_2 that satisfy a certain condition
  - definition $\Rightarrow R_1 \otimes_\Theta R_2 = \sigma_\Theta(R_1 \times R_2)$
  - equivalent SELECT statement

```
SELECT *
FROM R1 INNER JOIN R2 ON Θ
```

-

  - *natural join*
    - notation: $R_1 * R_2$
    - resulting relation:
      - schema: the union of the attributes of the two relations (attributes with the same name in R_1 and R_2 appear once in the result)
      - tuples: obtained from tuples <r_1, r_2>, where r_1 in R_1, r_2 in R_2, and r_1 and r_2 agree on the common attributes of R_1 and R_2
    - let $R_1[\alpha]$, $R_2[\beta]$, $\alpha \cap \beta = \{A_1, A_2, \ldots, A_m\}$; then:

$$R_1 * R_2 = \pi_{\alpha \cup \beta}(R_1 \otimes_{R_1.A_1 = R_2.A_1\ AND\ \ldots\ AND\ R_1.A_m = R_2.A_m} R_2)$$

    - equivalent SELECT statement

```
SELECT *
FROM R1 NATURAL JOIN R2
```

- *left outer join*
  - notation (in these notes): $R_1 \ltimes_C R_2$
  - resulting relation:
    - schema: the attributes of R_1 followed by the attributes of R_2
    - tuples: tuples from the condition join $R_1 \otimes_c R_2$ + the tuples in R_1 that were not used in $R_1 \otimes_c R_2$ combined with the *null* value for the attributes of R_2
  - equivalent SELECT statement

```
SELECT *
FROM R1 LEFT OUTER JOIN R2 ON C
```

- *right outer join*
  - notation: $R_1 \rtimes_C R_2$
  - resulting relation:
    - schema: the attributes of R_1 followed by the attributes of R_2
    - tuples: tuples from the condition join $R_1 \otimes_c R_2$ + the tuples in R_2 that were not used in $R_1 \otimes_c R_2$ combined with the *null* value for the attributes of R_1
  - equivalent SELECT statement

```
SELECT *
FROM R1 RIGHT OUTER JOIN R2 ON C
```

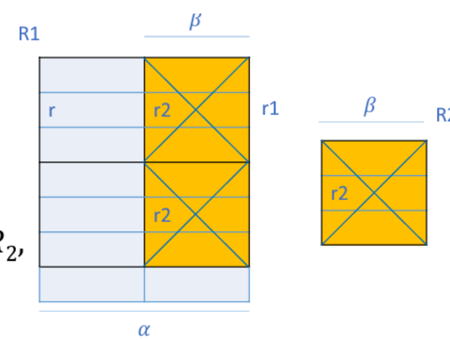
- *full outer join*
  - notation: $R_1 \bowtie_C R_2$
  - resulting relation:
    - schema: the attributes of R_1 followed by the attributes of R_2
    - tuples:
      - tuples from the condition join $R_1 \otimes_c R_2$ +
      - the tuples in R_1 that were not used in $R_1 \otimes_c R_2$ combined with the *null* value for the attributes of R_2 +
      - the tuples in R_2 that were not used in $R_1 \otimes_c R_2$ combined with the *null* value for the attributes of R_1
  - equivalent SELECT statement

```
SELECT *
FROM R1 FULL OUTER JOIN R2 ON C
```

- *left semi join*
  - notation: $R_1 \rhd R_2$
  - resulting relation:
    - schema: R_1's schema
    - tuples: the tuples in R_1 that are used in the natural join $R_1 * R_2$

- *right semi join*
  - notation: $R_1 \lhd R_2$
  - resulting relation:
    - schema: R_2's schema
    - tuples: the tuples in R_2 that are used in the natural join $R_1 * R_2$

- *division*
  - notation: $R_1 \div R_2$
  - $R_1[\alpha]$, $R_2[\beta]$, $\beta \subset \alpha$
  - resulting relation:
    - schema: $\alpha - \beta$
    - tuples: a record $r \in R_1 \div R_2$ iff $\forall r_2 \in R_2$, $\exists r_1 \in R_1$ such that:
      - $\pi_{\alpha-\beta}(r_1) = r$
      - $\pi_\beta(r_1) = r_2$
      - i.e., a record r belongs to the result if in R_1 r is concatenated with every record in R_2



- assignment

  $R[list] \coloneqq expression$

  - the expression's result (a relation) is assigned to a variable (R[list]), specifying the name of the relation [and the names of its columns]
- eliminating duplicates from a relation

  $\delta(R)$
- sorting records in a relation

  $S_{\{list\}}(R)$
- grouping

$$\gamma_{\{list1\}\ \text{group by}\ \{list2\}}(R)$$

  - R's records are grouped by the columns in *list2*
  - *list1* (that can contain aggregate functions) is evaluated for each group of records

Sabina S. CS

- independent set of operators

$$\{\sigma, \pi, \times, \cup, -\}$$

  - deriving the other operators

- the other operators are obtained as follows (some expressions have already been introduced):
  - $R_1 \cap R_2 = R_1 - (R_1 - R_2)$
  - $R_1 \otimes_C R_2 = \sigma_C(R_1 \times R_2)$

Sabina S. CS

- the other operators are obtained as follows (some expressions have already been introduced):
  - $R_1[\alpha]$, $R_2[\beta]$, $\alpha \cap \beta = \{A_1, A_2, \dots, A_m\}$, then:

    $R_1 * R_2 = \pi_{\alpha \cup \beta}(R_1 \otimes_{R_1.A_1 = R_2.A_1 \ AND \ \dots \ AND \ R_1.A_m = R_2.A_m} R_2)$

  - $R_1[\alpha]$, $R_2[\beta]$, $R_3[\beta] = \{(null, \dots, null)\}$, $R_4[\alpha] = \{(null, \dots, null)\}$

    $R_1 ⟕_C R_2 = (R_1 \otimes_c R_2) \cup (R_1 - \pi_\alpha(R_1 \otimes_c R_2)) \times R_3$

    $R_1 ⟖_C R_2 = (R_1 \otimes_c R_2) \cup R_4 \times (R_2 - \pi_\beta(R_1 \otimes_c R_2))$

    $R_1 ⟗_C R_2 = (R_1 ⟕_C R_2) \cup (R_1 ⟖_C R_2)$

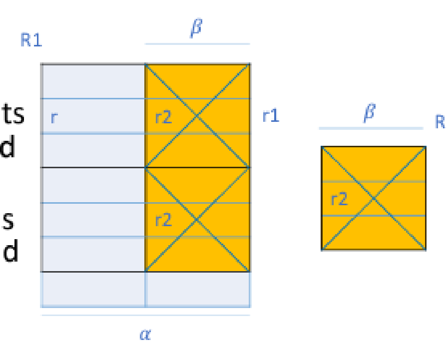
  - $R_1[\alpha]$, $R_2[\beta]$

    $R_1 \rhd R_2 = \pi_\alpha(R_1 * R_2)$

    $R_1 \lhd R_2 = \pi_\beta(R_1 * R_2)$

  - if $R_1[\alpha]$, $R_2[\beta]$, $\beta \subset \alpha$, then $r \in R_1 \div R_2$ iff $\forall r_2 \in R_2, \exists r_1 \in R_1$ such that: $\pi_{\alpha-\beta}(r_1) = r$ and $\pi_\beta(r_1) = r_2$

    => r is in $\pi_{\alpha-\beta}(R_1)$, but not all the elements in $\pi_{\alpha-\beta}(R_1)$ are in the result
  - $(\pi_{\alpha-\beta}(R_1)) \times R_2$ contains all the elements with one part in $\pi_{\alpha-\beta}(R_1)$ and the second part in R_2
  - to obtain values that are disqualified, R_1 is subtracted from the obtained relation, and the result is projected on $\alpha - \beta$
  - the final expression:



$$R_1 \div R_2 = \pi_{\alpha-\beta}(R_1) - \pi_{\alpha-\beta}((\pi_{\alpha-\beta}(R_1)) \times R_2 - R_1)$$

- query optimization
  - stamtement execution stages:
    - client: generate sql statement, send it to server
    - server:

- analyze sql statement syntatically
- translate statement into an internal form(relational algerbra expression)
- transform internal form into an optimal form
- generate a procedural execution plan
- evaluate procedural plan, send result to client

- evaluation
  - operands for relational operators
    - database tables
    - temporary tables obtained by evaluation some relational operators
  - several evaluation algorithms can be used for a relational algebra operator
- a join can be defined as a cross-product followed by a selection
- joins arise more often in practice than cross-products
- in general, the result of a corss product is much larger than the result of a join
- its important to impelement the join without materilizing the underlysing cross-product, by applying selections and projections as soon as possible
- Cross-join algotithm

  - this algorithm is used to evaluate a cross-product:
    - R CROSS JOIN S
    - R INNER JOIN S ON C (C evaluates to TRUE)
    - SELECT ... FROM R, S ..., no join condition between R and S
  - b_R, b_S
    - the number of blocks storing R and S, respectively
  - m, n
    - the number of blocks from R and S that can simultaneously appear in the main memory (there are m+n buffers for the 2 tables)

  - for every group of max m blocks in R:
    - read the group of blocks from R into the main memory; let M1 be the set of records in these blocks
    - for every group of max. n blocks in S:
      - read the group of blocks from S into main memory; let M2 be the set of records in these blocks
      - for every r E M1:
        - for every s E M2: add (r,s) to the result set
  - complexity

$$b_R + \left\lceil \frac{b_R}{m} \right\rceil * b_S \qquad (1)$$

(number of blocks in R; for every group of max. m blocks in R, read S)

- to minimize this value, m should be maximized (the other operands are constants); one buffer can be used for S (so n = 1), while the remaining space can be used for R (m max.)
- switch the 2 relations (in the algorithm and when computing the complexity) => complexity:

$$b_S + \left\lceil \frac{b_S}{n} \right\rceil * b_R \qquad (2)$$

- choose better version

- Nested Loops Join
  - The cross join alg can be used to evaluate a join between 2 tables
  - for every element (r,s) in the cross-product, evaluate the condition in the join operator
  - elements (r,s) that dont meet the join condition are eliminated
- Indexed Nested Loops Join
  - this algorithm is used to evaluate $R \otimes_C S$, where C ≡ (R.A=S.B), and there is an index on A (in R) or on B (in S)
  - in the algorithm description below, we assume there is an index on column B in table S
  - for every block in R:

- read the block into main memory: let M be the set of records in the block
- for every r E m:

```
determine v = πA(r)
use the index on B in S to determine records s ∈ S with
value v for B; for every such record s, the pair (r,s)
is added to the result
```

- merge join

  - this algorithm is used to evaluate $R \otimes_C S$, where C ≡ (R.A=S.B), and there are no indexes on A (in R) and B (in S)
  - `sort R and S on the columns used in the join: R on A, S on B`
  - `scan obtained tables; let r in R and s in S be 2 current records`
    - `if r.A = s.B: add (r', s') to the result; r' is in the set of all consecutive records in R with A = r.A, similarly for s' in S; next(r); next(s) (get a record with the next value for A and B)`
    - `if r.A < s.B: next(r) (determine record in sorted R with the next value for A)`
    - `if r.A > s.B: next(s) (determine record in sorted S with the next value for B)`

- hash join

  - this algorithm is used to evaluate $R \otimes_C S$, where C ≡ (R.A = S.B)

  1. partitioning phase

  - hash R and S on the join column, use the same hash function *h*

  => partitions

  2. probing phase

  - tuples in partition R_x are compared only with tuples in partition S_x (tuples in partition R_1 cannot join with tuples in partition S_2, for instance, as they have a different hash value)

- relational algebra equivalences

* $\sigma_C(\pi_\alpha(R)) = \pi_\alpha(\sigma_C(R))$

- selection reduces the number of records for projection; in the second expression, the projection operator analyzes fewer records
- optimization - algorithm that evaluates both operators in a single pass of R

* perform one pass instead of 2:

$$\sigma_{C1}(\sigma_{C2}(R)) = \sigma_{C1\ \mathrm{AND}\ C2}(R)$$

* replace cross-product and selection by condition join (a number of condition join algorithms don't evaluate the cross-product):

$$\sigma_C(R \times S) = R \otimes_C S$$

, where C - join condition between R and S

* R and S - compatible schemas:

$\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$
$\sigma_C(R \cap S) = \sigma_C(R) \cap \sigma_C(S)$
$\sigma_C(R - S) = \sigma_C(R) - \sigma_C(S)$

* $\sigma_C(R \times S)$

particular cases:

- C contains only attributes from R:

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

- C = C1 AND C2, C1 contains only attributes from R, C2 - only attributes from S:

$$\sigma_{C1\ \mathrm{AND}\ C2}(R \times S) = \sigma_{C1}(R) \times \sigma_{C2}(S)$$

- C = C1 AND C2, C2 - join condition between R and S:

$$\sigma_{C1\ \mathrm{AND}\ C2}(R \times S) = \sigma_{C1}(R \otimes_{C2} S)$$

Sabina S. CS

* $\pi_\alpha(R \cup S) = \pi_\alpha(R) \cup \pi_\alpha(S)$

* $\pi_\alpha(R \otimes_C S) = \pi_\alpha(\pi_{\alpha 1}(R) \otimes_C \pi_{\alpha 2}(S))$
• $\alpha 1$: attributes in R that appear in α or C
• $\alpha 2$: attributes in S that appear in α or C

* associativity and commutativity for some relational operators
• associativity and commutativity for ∪ and ∩
• associativity for the cross-product and the natural join
• "equivalent" results (same records, but different column order) when commuting operands in × and certain join operators
  • R × S = S × R – when using the Cross Join algorithm, the order of the data sources is important

Sabina S. CS

* transitivity of some relational operators for the join operators - additional filters could be applied before the join:
• (A>B AND B>3) ≡ (A>B AND B>3 AND A>3)
• example: A is in R, B is in S:

$$R \otimes_{A>B \text{ AND } B>3} S = (\sigma_{A>3}(R)) \otimes_{A>B} (\sigma_{B>3}(S))$$

• (A=B AND B=3) ≡ (A=B AND B=3 AND A=3)
• example: A is in R, B is in S:

$$R \otimes_{A=B \text{ AND } B=3} S = (\sigma_{A=3}(R)) \otimes_{A=B} (\sigma_{B=3}(S))$$

* evaluating $\sigma_C(R)$, where C ≡ $(R.A \in \delta(\pi_{\{B\}}(S)))$; avoid evaluating C for every record of R; the initial evaluation is equivalent to:

$$R \otimes_{R.A=S.B} (\delta(\pi_{\{B\}}(S)))$$