CS186 Midterm 1 Cheat Sheet

```
SELECT  S.sname
FROM    Sailors S
WHERE EXISTS
      (SELECT  *
       FROM  Reserves R
       WHERE R.bid=102 AND S.sid=R.sid)
```

<u>SQL:</u>

- Pros: Declarative and implemented widely, general-purpose and feature-rich
- Cons: Constrained, not targeted at turning complete tasks
- **DDL**: data definition language, used to define and modify schema
- **DML:** used to write intuitive queries
- **Primary key** provides unique key lookup, cannot have duplicate values and can contain more than 1 column (firstname, lastname)

```
SELECT [DISTINCT] <column
  FROM <single table>
[WHERE <predicate>]
[GROUP BY <column list>
  [HAVING <predicate>] ]
[ORDER BY <column list>]
[LIMIT <integer>];
```

- **Foreign key** references a table via another table's primary key
  - **FROM** compute cross product of tables.
  - **WHERE** Check conditions, discard fails
  - **SELECT** Specify desired fields in output.
  - **DISTINCT** (optional) eliminate duplicate rows.
- Nested Queries in WHERE clause:
  - Keywords: **IN**, **EXISTS**, **NOT IN**, **NOT EXISTS**, **op ANY**, **op ALL** (ex. < ANY)

```
SELECT *
FROM   Sailors S
WHERE  S.rating >= ALL
   (SELECT  S2.rating
      FROM  Sailors S2)
```

```
SELECT *
FROM   Sailors S
WHERE  S.rating =
(SELECT  MAX(S2.rating)
   FROM  Sailors S2)
```

- These implementations of **ARGMAX** that gives us highest rated sailor

```
SELECT *
FROM   Sailors S
WHERE  S.rating >= ALL
   (SELECT  S2.rating
      FROM  Sailors S2)
```

```
SELECT  *
FROM    Sailors S
ORDER BY rating DESC
LIMIT 1;
```

- These implementations of **ARGMAX** that gives us highest rated sailor but are slightly different. Right gives us 1, left gives us all highest
- **Inner/Natural Join** takes cross product (think 61a)
- **Left Outer Join** preserves all rows on the left.
  - **Ex.** FROM sailors2 s LEFT OUTER JOIN reserves2 r ON s.sid = r.sid, sailors2 table preserved
- **Right Outer Join** not used as much, but opposite of left outer join
- **Full Outer Join** preserves both left and right table
- **Views** are just sub queries.
  - Syntax: *CREATE VIEW viewname AS SELECT*
- **With tables** are on the fly views (cs61a)

<u>Set Semantics & RegEx</u>:

- **UNION ALL:** sum of cardinalities {A, B, C} UNION ALL {B, C} = {A, B, B, C, C}
- **INTERSECT ALL:** min of all cardinalities {A, B, C} INTERSECT ALL {B, C} = {B, C}
- **EXCEPT ALL:** difference of cardinalities {A, B, C} EXCEPT ALL {B, C, D} = {A}

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that starts with "a"
WHERE CustomerName LIKE '%a'	Finds any values that ends with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%_%'	Finds any values that starts with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that starts with "a" and ends with "o"

<u>Storing Data: Disks, Buffers:</u>

- DBMS organized in layers, each layer abstracts the layer below. **Concurrency Control** is between relational operators and files and index management while **Recovery** is at disk space management level
- **Hierarchy:** files -> blocks/pages -> records
  - Tables stored as **logical files** consisting of **pages** each containing a collection of **records**
- Pages are managed on **disk** by the **disk space manager** and in **memory** by the **buffer manager**
- Components of a **disk:**
  - **Platters**: physical disks one on top of the other
  - **Arm assembly:** one arm per platter: read/write to that platter
  - **Tracks:** circles on the platter, different distance from centre
  - **Cylinder:** tracks with same radii on all platters
  - **Sectors:** part of a track, an arc of data, fixed number of bits
- Disk**: Seek time** moving arms to track position (2-3ms) **rotational delay** (0-4ms) **transfer time** transfer from disk surface(0.25ms)
- **Flash** is much faster, limited # of writes
- **Disk space management** provides API to read and write pages down
- **Buffer management** tells DSM to read/write/allocate pages

- A **DB File** is a collection of pages (insert/delete/modify/fetch/scan)
- **Unordered Heap Files** records placed arbitrarily across pages
  - Pages contains 2 pointers (header points free and full pages)
  - **Page directory** includes #free bytes.
- A **page** stores records, **page header** may store # records, free space, next/last pointer, bitmaps, slot table.
  - **Fixed length** records can be packed closely, but deletion leaves holes. Use **bitmap** to says where the holes are
  - **Variable length** records use a **footer** to keep track of length + pointer to beginning of record (reverse order) and pointer to free space. **Record Id is pageId + slotId** reshuffle if you want
- A **record** (stores a row on a table) should save space and be fast to access any field.
  - Using **fixed length** for each record is bad, lots of wasted space and variable length records could be much bigger
  - **Delimiters** (to separate fields) bad idea, what if in field?
  - Adding **length** before field – scan takes time
  - Add a **record header** with pointers to var length fields

<u>File Organizations:</u>

- **Heap files**: good for full scan of all records
- **Sorted files**: good for ordered retrieval or range retrieval
- **Clustered files & indexes**: good for fast lookup & modification
- **B** = # of blocks/file, **R** = # record/block, **D** = time to read/write block

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	1.5*B*D
Equality Search	0.5*B*D	$(\log_2 B)*D$	$(\log_F 1.5*B + 1)*D$
Range Search	B*D	$((\log_2 B)+pages))*D$	$((\log_F 1.5*B)+pages)*D$
Insert	2*D	$((\log_2 B)+B)*D$	$((\log_F 1.5*B)+2)*D$
Delete	(0.5*B+1) * D	$((\log_2 B)+B)*D$	$((\log_F 1.5*B)+2)*D$

<u>Tree Indexes:</u>

- An **index** is data structure that enables fast **lookup** of **data entries** by **search key**
  - **Lookup:** may support different operations (equality, range…)
  - **Search Key:** any subset of columns (firstname, lastname)
  - **Data Entries:** items stored in the index (k, recordId)
  - **Many Types:** B+-Tree, Hash, R-Tree, GiST
- **ISAM** leaves spaces in the leaves to allow for inserts (like B+ trees)
  - Pages physically sorted in logical order (no next pointers)
  - **Positives** (given no overflow)**:** 1-record lookup by search key, scan all records, scan a range of records by search key
  - **Negatives:** No balancing after insertion (overflow pages)
- **B+-Tree** is similar to ISAM (close structure) two major differences
  - **Dynamic Tree Index** make it always balanced (no overflow)
  - B trees store data entries in **leaves only**.
  - **Positives:** 1-rec lookup by search key, balance after insertion
  - **Uncertainties:** Scan all records (not sequentially stored), scan a range of records by search key (same argument)
  - Deletion is tricky (let pages slowly go empty, and delete)
- **B+-Tree bulk loading** lets us create all the leaves first, then build the B+-Tree from the leaves – Positives: fewer ios, control "fill factors"
- Searching a range is sequentially only if we do < or > on the last item:
  - a = 31 & s = 40, a = 55 & s > 200, a > 31 & s = 400
- Alt 1 store record content on B+ Tree roots (big tree, but no pointers)
- Alt 2 (clustered index) leaves = [key, recordid] (record stored else)
- Alt 3 is just alt 2 but [key, [recordids]] list of record ids
- **Clustered** means that the records themselves are stored in the same order as the leaves of the B tree, and **unclustered** is not
- For variable length keys and records, use bytes per node (not entries)

<u>Buffer Management:</u>

- **Files and Index Management** tells buffer management what to do.
  - Bytes are manipulated at buffer level (can't do disk)
  - **Dirty bit:** page in the buffer pool need to be written to disk
  - **Pin count:** tells us how many processes are using the page
  - If pool is full, always replace an **un-pinned** page in the pool
    - Pin the page and return its address
- **LRU (Least Recently Used):** replaces least recently used page
  - Need to keep track of **last used counter** (find min-costly)
- **Clock Policy:** replace first non-pinned, non-ref bit active page
  - Needs a **clock hand** and a **ref bit**
- **MRU (Most recently Used):** replaces most recently used page

<u>Sorting and Hashing:</u>

- **Sorting**: Good for eliminating duplicates, grouping and ordering
- **Single pass streaming** forces us to minimize RAM, but we want to call read/write rarely. Computer f(x) for each record and write result
- **2-way sort** requires 3 buffer pages: input buffer takes 2, outputs 1
- **External Merge Sort** takes B buffer pages, requires ceil(N / B) runs to sort N pages. <u>Passes = 1 + ceil(logB-1(ceil(N/B))) Cost = 2N * # passes</u>
  - B*(B-1) for 2 passes, 1 pass streaming for merge