Introduction to Database - Final Project

Goals

Implement an on-disk DBMS (Database Management System)

Perform the following query on a set of .csv files:

```
SELECT AVG(ArrDelay) FROM ontime WHERE Origin = {QueryAirport1} AND Dest =
{QueryAirport2};
```

Datasets

Data expo '09, ASA Sections on Statistical Computing

- 2006.csv
- 2007.csv
- 2008.csv

21604865 rows (21134119 valid), **29** columns

Requirements

- Test Environment: 2GB RAM, gcc 4.8.4, Ubuntu 14.04 LTS
- No custom makefiles
- No parallelization
- No pre-calculations
- No data can be stored in memory before queries

Installation

The GitHub repo will be available after the deadline.

Dependencies:

- g++ 4.8.4 or newer
- make

Tested on `Windows`, `Mac OSX`, `Ubuntu` and `FreeBSD`

```
# Clone the repo
git clone https://github.com/lnishan/dbhw4.git
cd dbhw4

# Download dataset
# Prerequisites: wget, bzip2
./download.sh

# Build and run
# O0 - O3 are all supported
# O0, if no argument is passed in
./run.sh O2

# Build, run and organize results
# O0 - O3 are all supported
# O0, if no argument is passed in
./test.sh O2
```

Implementations/Optimizations

Basic Data Structures

- A specialized and highly-optimized `hash map`
  ➔ Simple tabulation hashing (a fast and great type of hash function)
  ➔ Linear probing (benefits from spatial locality/hardware prefetching)
  ➔ A generalized version is available at: https://github.com/lnishan/Notebook/blob/master/Source/BasicDS_Hashmap_Unordered%20Map_umap.cpp

- A self-implemented and faster `vector`
  ➔ No overheads induced by "good" design practices
  ➔ Significantly less memory reallocations due to more redundancies
  ➔ At least 2x faster than std::vector on insertions
  ➔ A almost-full-standard-compliant vector is available at: https://github.com/lnishan/vector

- An extra-large `read buffer`
  ➔ Applicable to both import and indexing
  ➔ Reduce system call overheads

- An extra-large `write buffer`
  ➔ No need to write to the file before queries
  ➔ Reduce system call overheads

Import

- Compact and uniform file format
  ➔ Simple decoding to boost input efficiency

Generated database file (temp/lnishan.db):

```
ATLPHX6
ATLPHX-5
ATLPHX-23
AUSPHX-8
AUSPHX0
BDLCLT2
BDLCLT20
BDLCLT7
BDLCLT-7
BDLCLT-4
```

Explanation:

- Byte [0 - 2]: Origin
- Byte [3 - 5]: Dest
- Byte [6 -]: ArrDelay

Indexing

Algorithm:

1. Read database file
2. While reading the file, insert the position of each entry into the hash map
   eg. after insert, hash[0] = {0, 100, 200, 500}, hash[1] = {40, 320}, … etc.
3. Generate a new index file (temp/index) by condensing the database file in the order of the hash table buckets.
4. While generating the index file, record the starting position of each hash table entry.

Other Optimizations

- Reduce branch instructions
  ➔ Eliminate branch misprediction penalties

- High reserves on vectors
  ➔ Reduce memory reallocations
  ➔ Trade memory for speed

- Use mostly C library
  ➔ C library is oftentimes more efficient
  ➔ Some C++ STLs are incredibly slow, especially std::string and related I/O functions

- C++11 features
  ➔ Use newer functions with underlying move semantics to prevent unnecessary data copying

- Generally robust codes throughout

Results

Here is the screenshot of the result using the provided main.cpp and makefile.

```
Time taken for import: 10.52s
Time taken for creating index: 12.19s
Time taken for making queries: 0.00s
```

Here are the screenshots of the results using my modified main.cpp, makefile and test script.

```
#1
Time taken for import: 9.1382s
Time taken for queries (pre-indexing): 2.0250s
Time taken for indexing: 11.4494s
Time taken for queries: 0.0012s
Results (pre-indexing): 17.2076 11.4790 11.2612 12.6459 7.1969
Results: 17.2076 11.4790 11.2612 12.6459 7.1969
```

```
#2
Time taken for import: 9.1390s
Time taken for queries (pre-indexing): 2.0495s
Time taken for indexing: 11.7940s
Time taken for queries: 0.0012s
Results (pre-indexing): 17.2076 11.4790 11.2612 12.6459 7.1969
Results: 17.2076 11.4790 11.2612 12.6459 7.1969
```

```
#3
Time taken for import: 9.3398s
Time taken for queries (pre-indexing): 2.0643s
Time taken for indexing: 11.6678s
Time taken for queries: 0.0013s
Results (pre-indexing): 17.2076 11.4790 11.2612 12.6459 7.1969
Results: 17.2076 11.4790 11.2612 12.6459 7.1969
```

```
#4
Time taken for import: 9.2910s
Time taken for queries (pre-indexing): 2.0059s
Time taken for indexing: 11.6149s
Time taken for queries: 0.0015s
Results (pre-indexing): 17.2076 11.4790 11.2612 12.6459 7.1969
Results: 17.2076 11.4790 11.2612 12.6459 7.1969
```

```
#5
Time taken for import: 9.2460s
Time taken for queries (pre-indexing): 1.9712s
Time taken for indexing: 12.2713s
Time taken for queries: 0.0011s
Results (pre-indexing): 17.2076 11.4790 11.2612 12.6459 7.1969
Results: 17.2076 11.4790 11.2612 12.6459 7.1969
```

```
| # | Import | Dry-Queries | Indexing | Queries |
| --- | --- | --- | --- | --- |
| 1 | 9.1382 | 2.0250 | 11.4494 | 0.0012 |
| 2 | 9.1390 | 2.0495 | 11.7940 | 0.0012 |
| 3 | 9.3398 | 2.0643 | 11.6678 | 0.0013 |
| 4 | 9.2910 | 2.0059 | 11.6149 | 0.0015 |
| 5 | 9.2460 | 1.9712 | 12.2713 | 0.0011 |
| **Avg.** | **9.23080** | **2.02318** | **11.75948** | **0.00126** |
```

Origin	Dest	Avg. ArrDelay	Query Time (Pre-indexing)	Index Time	Query Time (Post-indexing)
IAH	JFK	17.2076	2.02318	11.75948	0.00126
IAH	LAX	11.4790	↑	↑	↑
JFK	LAX	11.2612	↑	↑	↑
JFK	IAH	12.6459	↑	↑	↑
LAX	IAH	7.1969	↑	↑	↑

Cross-platform Results

Date: 1:16 PM (UTC+8), June 13, 2016
Hardware: 50GB Vmware Disk, 4GB RAM, 1 vCore (Guest VM, Host: 1TB SSHD, 16GB RAM, i7-3770)
Environment: gcc 6.1.1, Kubuntu 16.04 LTS

#	Import	Dry-Queries	Indexing	Queries
1	3.9336	1.1558	6.0963	0.0007
2	3.8830	1.1459	6.3977	0.0007
3	3.8818	1.1449	5.9865	0.0007
4	3.8821	1.1487	6.0527	0.0006
5	3.8586	1.1467	6.0655	0.0006
Avg.	**3.88782**	**1.14840**	**6.11974**	**0.00066**

Date: 9:55 PM (UTC+8), June 13, 2016
Hardware: 30GB Vmware Disk, 4GB RAM, 1 vCore (Guest VM, Host: 1TB SSHD, 16GB RAM, i7-3770)
Environment: gcc 6.1.1, FreeBSD 10.3-RELEASE

#	Import	Dry-Queries	Indexing	Queries
1	4.2109	1.1797	11.5312	0.0000

#	Import	Dry-Queries	Indexing	Queries
2	4.1797	1.1797	11.4609	0.0000
3	4.1172	1.1797	11.3438	0.0000
4	4.1797	1.2031	11.4688	0.0000
5	4.1875	1.1875	11.4297	0.0078
Avg.	**4.17500**	**1.18594**	**11.44688**	**0.00156**

Date: 1:45 AM (UTC+8), June 10, 2016
Hardware: 16GB Persistent Disk, 3.75GB RAM, 1 HyperThread on 2.5GHz Xeon E5 v2 (Google Compute Engine n1-standard-1)
Environment: gcc 6.1.1, Ubuntu 16.04 LTS

#	Import	Dry-Queries	Indexing	Queries
1	5.4656	1.6126	10.2230	0.0009
2	5.7495	1.5904	9.6879	0.0009
3	5.8680	1.6061	10.0110	0.0009
4	5.5988	1.5938	9.6082	0.0009

#	Import	Dry-Queries	Indexing	Queries
5	5.6484	1.5993	9.8001	0.0009
Avg.	**5.66606**	**1.60044**	**9.86604**	**0.00090**

Date: 3:15 AM (UTC+8), June 15, 2016
Hardware: 40GB SSD, 2GB RAM, 1 CPU on DigitalOcean VPS
Environment: gcc 6.1.1, Ubuntu 16.04 LTS

#	Import	Dry-Queries	Indexing	Queries
1	9.4570	1.9862	11.7226	0.0011
2	9.5078	1.9648	11.6216	0.0015
3	9.3291	1.9824	11.5000	0.0012
4	9.3116	1.9714	11.8074	0.0013
5	9.6504	2.1179	12.2284	0.0012
Avg.	**9.45118**	**2.00454**	**11.77600**	**0.00126**

Date: 9:49 PM (UTC+8), June 13, 2016
Hardware: 1TB SSHD, 16GB RAM, i7-3770 (Desktop)
Environment: Cygwin-gcc 5.3.0, Windows 10 Enterprise 64-bit

#	Import	Dry-Queries	Indexing	Queries
1	4.0320	1.2190	20.1400	0.0000
2	4.0320	1.1870	20.3600	0.0000
3	4.0470	1.2030	20.2660	0.0000
4	4.0160	1.2030	20.1870	0.0000
5	4.0940	1.2180	20.5640	0.0000
Avg.	**4.04420**	**1.20600**	**20.30340**	**0.00000**

Date: 7:05 PM (UTC+8), June 13, 2016
Hardware: 256GB SSD, 16GB RAM, i5-4260U (MacBook Air Early 2014)
Environment: gcc 6.1.0, OS X El Capitan

#	Import	Dry-Queries	Indexing	Queries
1	6.4050	1.8362	15.8634	0.0015
2	6.4097	1.8452	15.9725	0.0017

#	Import	Dry-Queries	Indexing	Queries
3	6.3648	1.8423	16.0777	0.0017
4	6.3082	1.8224	15.8907	0.0017
5	6.3190	1.8200	15.8589	0.0018
Avg.	**6.36134**	**1.83322**	**15.93264**	**0.00168**

Date: 10:09 PM (UTC+8), June 13, 2016
Hardware: 1TB SSHD, 16GB RAM, i7-3770 (Desktop)
Environment: Visual Studio 2015, Windows 10 Enterprise 64-bit

#	Import	Dry-Queries	Indexing	Queries
1	5.3200	1.7310	27.5740	0.0000
2	5.3160	1.7110	27.2360	0.0000
3	5.2600	1.7040	27.1780	0.0000
4	5.2810	1.7050	27.1600	0.0000
5	5.2780	1.7100	27.1940	0.0010
Avg.	**5.29100**	**1.71220**	**27.26840**	**0.00020**