TODAY’S AGENDA

In-Memory Data Layout
Storage Models
Project #2: Performance Profiling
DATA ORGANIZATION

Index

Fixed-Length Data Blocks

Variable-Length Data Blocks

Memory Address
One can think of an in-memory database as just a large array of bytes. The schema tells the DBMS how to convert the bytes into the appropriate type.

Each tuple is prefixed with a header that contains its meta-data.

Storing tuples with just their fixed-length data makes it easy to compute the starting point of any tuple.
DATA REPRESENTATION

INTEGER/BIGINT/SMALLINT/TINYINT
→ C/C++ Representation

NUMERIC
→ IEEE-754 Standard

VARCHAR/VARBINARY/TEXT/BLOB
→ Pointer to other location if type is ≥64-bits
→ Header with length and address to next location (if segmented), followed by data bytes.

TIME/DATE/TIMESTAMP
→ 32/64-bit integer of (micro)seconds since Unix epoch
CREATE TABLE JoySux ( 
  id INT PRIMARY KEY, 
  value BIGINT 
);
CREATE TABLE JoySux (id INT PRIMARY KEY, value BIGINT);

char[]

header id value
CREATE TABLE JoySux (id INT PRIMARY KEY, value BIGINT);

```sql
CREATE TABLE JoySux (id INT PRIMARY KEY, value BIGINT);
```

**char**

<table>
<thead>
<tr>
<th>header</th>
<th>id</th>
<th>value</th>
</tr>
</thead>
</table>

**Diagram:**

- **Header**: id
- **Value**: bigint
CREATE TABLE JoySux (
  id INT PRIMARY KEY,
  value BIGINT
);

char[]

header

id

value
CREATE TABLE JoySux (id INT PRIMARY KEY, value BIGINT);

char[] header id value

reinterpret_cast<int32_t*>

(address)
NULL DATA TYPES

Choice #1: Special Values
→ Designate a value to represent NULL for a particular data type (e.g., INT32_MIN).

Choice #2: Null Column Bitmap Header
→ Store a bitmap in the tuple header that specifies what attributes are null.

Choice #3: Per Attribute Null Flag
→ Store a flag that marks that a value is null.
→ Have to use more space than just a single bit because this messes up with word alignment.
# NULL DATA TYPES

## Integer Numbers

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Size (Not Null)</th>
<th>Synonyms</th>
<th>Min Value</th>
<th>Max Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>2 bytes</td>
<td>1 byte</td>
<td>BOOLEAN</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>BIT</td>
<td>9 bytes</td>
<td>8 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TINYINT</td>
<td>2 bytes</td>
<td>1 byte</td>
<td></td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>4 bytes</td>
<td>2 bytes</td>
<td></td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>MEDIUMINT</td>
<td>4 bytes</td>
<td>3 bytes</td>
<td></td>
<td>-8388608</td>
<td>8388607</td>
</tr>
<tr>
<td>INT</td>
<td>8 bytes</td>
<td>4 bytes</td>
<td>INTEGER</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>BIGINT</td>
<td>12 bytes</td>
<td>8 bytes</td>
<td></td>
<td>-2 ** 63</td>
<td>(2 ** 63) - 1</td>
</tr>
</tbody>
</table>
NULL DATA TYPES

Integer Numbers

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Size (Not Null)</th>
<th>Synonyms</th>
<th>Min Value</th>
<th>Max Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>2 bytes</td>
<td>1 byte</td>
<td>BOOLEAN</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>BIT</td>
<td>9 bytes</td>
<td>8 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TINYINT</td>
<td>2 bytes</td>
<td>1 byte</td>
<td></td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>4 bytes</td>
<td>2 bytes</td>
<td></td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>MEDIUMINT</td>
<td>4 bytes</td>
<td>3 bytes</td>
<td></td>
<td>-8388608</td>
<td>8388607</td>
</tr>
<tr>
<td>INT</td>
<td>8 bytes</td>
<td>4 bytes</td>
<td>INTEGER</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>BIGINT</td>
<td>12 bytes</td>
<td>8 bytes</td>
<td></td>
<td>-2 ** 63</td>
<td>(2 ** 63) - 1</td>
</tr>
</tbody>
</table>

this messes up with word alignment.
NULL DATA TYPES

Choice #1: Special Values
→ Designate a value to represent NULL for a particular data type (e.g., INT32_MIN).

Choice #2: Null Column Bitmap Header
→ Store a bitmap in the tuple header that specifies what attributes are null.

Choice #3: Per Attribute Null Flag
→ Store a flag that marks that a value is null.
→ Have to use more space than just a single bit because this messes up with word alignment.
The truth is that you only need to worry about word-alignment for cache lines (e.g., 64 bytes).

I’m going to show you the basic idea using 64-bit words since it’s easier to see...
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```
CREATE TABLE JoySux (
    id INT PRIMARY KEY,
    cdate TIMESTAMP,
    color CHAR(2),
    zipcode INT
);
```

`char[]`
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE JoySux (id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT);

char[]
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```sql
CREATE TABLE JoySux (
    id INT PRIMARY KEY,
    cdate TIMESTAMP,
    color CHAR(2),
    zipcode INT
);
```

*char[]*

64-bit Word 64-bit Word 64-bit Word 64-bit Word
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```
CREATE TABLE JoySux ( id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT );
```

```
32-bits
```

```
64-bit Word 64-bit Word 64-bit Word 64-bit Word
```

```
char[]
```

```
64-bit Word 64-bit Word 64-bit Word 64-bit Word
```
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE JoySux (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);

char[]

id  cdate

64-bit Word  64-bit Word  64-bit Word  64-bit Word
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE JoySux (id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT);

```
<table>
<thead>
<tr>
<th>id</th>
<th>cdate</th>
<th>color</th>
<th>zipcode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

`char[]`

64-bit Word 64-bit Word 64-bit Word 64-bit Word
All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE JoySux (  
id INT PRIMARY KEY,  
cdate TIMESTAMP,  
color CHAR(2),  
zipcode INT  
);
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE JoySux (id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT);

```
class JoySux {  
  int id;  
  java.util.Date cdate;  
  char color;  
  int zipcode;  
}
```
WORD-ALIGNED TUPLES

If the CPU fetches a 64-bit value that is not word-aligned, it has four choices:
→ Execute two reads to load the appropriate parts of the data word and reassemble them.
→ Read some unexpected combination of bytes assembled into a 64-bit word.
→ Throw an exception.
WORD-ALIGNED TUPLES

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

CREATE TABLE JoySux (id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT);

```sql
CREATE TABLE JoySux (id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), zipcode INT);
```

![Diagram showing word alignment in tuples](image-url)
**WORD-ALIGNED TUPLES**

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

```sql
CREATE TABLE JoySux (
    id INT PRIMARY KEY,
    cdate TIMESTAMP,
    color CHAR(2),
    zipcode INT
);
```

![Diagram of word alignment]

- **32-bits**
- **64-bits**
- **16-bits**
- **32-bits**
STORAGE MODELS

N-ary Storage Model (NSM)
Decomposition Storage Model (DSM)
Hybrid Storage Model
N-ARY STORAGE MODEL (NSM)

The DBMS stores all of the attributes for a single tuple contiguously.

Ideal for OLTP workloads where txns tend to operate only on an individual entity and insert-heavy workloads.

Use the tuple-at-a-time iterator model.
**NSM PHYSICAL STORAGE**

**Choice #1: Heap-Organized Tables**
→ Tuples are stored in blocks called a heap.
→ The heap does not necessarily define an order.

**Choice #2: Index-Organized Tables**
→ Tuples are stored in the index itself.
→ Not quite the same as a clustered index.
CLUSTERED INDEXES

The table is stored in the sort order specified by the primary key.
→ Can be either heap- or index-organized storage.

Some DBMSs always use a clustered index.
→ If a table doesn’t include a pkey, the DBMS will automatically make a hidden row id pkey.

Other DBMSs cannot use them at all.
→ A clustered index is non-practical in a MVCC DBMS using the Insert Method.
N-ARY STORAGE MODEL (NSM)

Advantages
→ Fast inserts, updates, and deletes.
→ Good for queries that need the entire tuple.
→ Can use index-oriented physical storage.

Disadvantages
→ Not good for scanning large portions of the table and/or a subset of the attributes.
The DBMS stores a single attribute for all tuples contiguously in a block of data. → Sometimes also called *vertical partitioning*.

Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table’s attributes.

Use the vector-at-a-time iterator model.
DECOMPOSITION STORAGE MODEL (DSM)

1970s: Cantor DBMS
1980s: DSM Proposal
1990s: SybaseIQ (in-memory only)
2000s: Vertica, Vectorwise, MonetDB
2010s: “The Big Three”
        Cloudera Impala, Amazon Redshift,
        SAP HANA, MemSQL
CLUSTERED INDEXES

Some columnar DBMSs store data in sorted order to maximize compression.
→ Bitmap indexes with RLE from last class

Vertica does not even use indexes because all columns are sorted.
**TUPLE IDENTIFICATION**

**Choice #1: Fixed-length Offsets**
→ Each value is the same length for an attribute.

**Choice #2: Embedded Tuple Ids**
→ Each value is stored with its tuple id in a column.

**Offsets**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

**Embedded Ids**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
DECOMPOSITION STORAGE MODEL (DSM)

Advantages
→ Reduces the amount wasted work because the DBMS only reads the data that it needs.
→ Better compression (last lecture).

Disadvantages
→ Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching.
OBSERVATION

Data is “hot” when first entered into database
→ A newly inserted tuple is more likely to be updated again the near future.

As a tuple ages, it is updated less frequently.
→ At some point, a tuple is only accessed in read-only queries along with other tuples.

What if we want to use this data to make decisions that affect new txns?
BIFURCATED ENVIRONMENT

OLTP Data Silos

OLAP Data Warehouse
BIFURCATED ENVIRONMENT

OLTP Data Silos

Extract
Transform
Load

OLAP Data Warehouse
OLTP Data Silos

Extract
Transform
Load

OLAP Data Warehouse
BIFURCATED ENVIRONMENT

OLTP Data Silos

Extract
Transform
Load

OLAP Data Warehouse
BIFURCATED ENVIRONMENT

OLTP Data Silos ➔ Extract ➔ Transform ➔ Load ➔ OLAP Data Warehouse
BIFURCATED ENVIRONMENT

OLTP Data Silos → Extract → Transform → Load → OLAP Data Warehouse
HYBRID STORAGE MODEL

Single logical database instance that uses different storage models for hot and cold data.

Store new data in NSM for fast OLTP
Migrate data to DSM for more efficient OLAP
HYBRID STORAGE MODEL

Choice #1: Separate Execution Engines
→ Use separate execution engines that are optimized for either NSM or DSM databases.

Choice #2: Single, Flexible Architecture
→ Use single execution engine that is able to efficiently operate on both NSM and DSM databases.
SEPARATE EXECUTION ENGINES

Run separate “internal” DBMSs that each only operate on DSM or NSM data.
→ Need to combine query results from both engines to appear as a single logical database to the application.
→ Have to use a synchronization method (e.g., 2PC) if a txn spans execution engines.

Two approaches to do this:
→ Fractured Mirrors (Oracle, IBM)
→ Delta Store (SAP HANA)
A CASE FOR FRACTURED MIRRORS
VLDB 2002

FRACTURED MIRRORS

Store a second copy of the database in a DSM layout that is automatically updated.
→ All updates are first entered in NSM then eventually copied into DSM mirror.
Store a second copy of the database in a DSM layout that is automatically updated.
→ All updates are first entered in NSM then eventually copied into DSM mirror.
FRACTURED MIRRORS

Store a second copy of the database in a DSM layout that is automatically updated.
→ All updates are first entered in NSM then eventually copied into DSM mirror.

A CASE FOR FRACTURED MIRRORS
VLDB 2002
FRACTURED MIRRORS

Store a second copy of the database in a DSM layout that is automatically updated.
→ All updates are first entered in NSM then eventually copied into DSM mirror.
FRACTURED MIRRORS

Store a second copy of the database in a DSM layout that is automatically updated.
→ All updates are first entered in NSM then eventually copied into DSM mirror.
DELTA STORE

Stage updates to the database in an NSM table. A background thread migrates updates from delta store and applies them to DSM data.
Stage updates to the database in an NSM table. A background thread migrates updates from delta store and applies them to DSM data.
Stage updates to the database in an NSM table. A background thread migrates updates from delta store and applies them to DSM data.
SINGLE, FLEXIBLE ARCHITECTURE

Use a single execution engine architecture that is able to operate on both NSM and DSM data.
→ Don’t need to store two copies of the database.
→ Don’t need to sync multiple database segments.

Note that a DBMS can use the delta-store for NSM data with a single architecture.
H2O ADAPTIVE STORAGE

Examine the access patterns of queries and then dynamically reconfigure the database to optimize decomposition and layout.

Copies columns into a new layout that is optimized for each query.
→ Think of it like a mini fractured mirror.
→ Use query compilation to speed up operations.
Original Data

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
</table>

**UPDATE** JoyStillSux
**SET** B = 1234
**WHERE** C = "xxx"

**SELECT** AVG(B)
**FROM** JoyStillSux
**WHERE** C = "yyy"

**SELECT** SUM(A)
**FROM** JoyStillSux
Original Data

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UPDATE JoyStillSux
SET B = 1234
WHERE C = “xxx”

SELECT AVG B
FROM JoyStillSux
WHERE C = “yyy”

SELECT SUM(A)
FROM JoyStillSux
**H₂O ADAPTIVE STORAGE**

**Original Data**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **UPDATE** JoyStillSux
  - SET \( B = 1234 \)
  - WHERE \( C = "xxx" \)

- **SELECT AVG** \( B \)
  - FROM JoyStillSux
  - WHERE \( C = "yyy" \)

- **SELECT SUM** \( A \)
  - FROM JoyStillSux
H2O ADAPTIVE STORAGE

**Original Data**

```
UPDATE JoyStillSux
SET B = 1234
WHERE C = "xxx"

SELECT AVG B
FROM JoyStillSux
WHERE C = "yyy"

SELECT SUM A
FROM JoyStillSux
```

**Adapted Data**

```
A B C D
```

```
A B C D
```
H₂O ADAPTIVE STORAGE

This approach is unable to handle updates to the database. It also unable to store tuples in the same table in a different layout.

This is because they are missing the ability to categorize whether data is hot or cold...
PELOTON ADAPTIVE STORAGE

**Original Data**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SQL Queries**

- **UPDATE** JoyStillSux
  - `SET B = 1234`
  - `WHERE C = "xxx"`

- **SELECT AVG(B)**
  - `FROM JoyStillSux`
  - `WHERE C = "yyy"`

- **SELECT SUM(A)**
  - `FROM JoyStillSux`
UPDATE JoyStillSux
  SET B = 1234
  WHERE C = "xxx"

SELECT AVG(B)
  FROM JoyStillSux
  WHERE C = "yyy"

SELECT SUM(A)
  FROM JoyStillSux
SELECT AVG(B)  
FROM JoyStillSux  
WHERE C = "yyy"

SELECT SUM(A)  
FROM JoyStillSux

UPDATE JoyStillSux  
SET B = 1234  
WHERE C = "xxx"

Original Data

Hot

Cold
**PELOTON ADAPTIVE STORAGE**

**Original Data**

```
SELECT AVG(B)
FROM JoyStillSux
WHERE C = “yyy”

UPDATE JoyStillSux
SET B = 1234
WHERE C = “xxx”

SELECT SUM(A)
FROM JoyStillSux
```

**Adapted Data**

- Hot
- Cold

```
A B C D
```

```
A B C D
```

```
A B C D
```

```
A B C D
```
PELOTON ADAPTIVE STORAGE

**Original Data**

```
UPDATE JoyStillSux
SET B = 1234
WHERE C = "yyy"

SELECT AVG(B)
FROM JoyStillSux
WHERE C = "yyy"

SELECT SUM(A)
FROM JoyStillSux
```

**Adapted Data**
CATEGORIZING DATA

Choice #1: Manual Approach
→ DBA specifies what tables should be stored as DSM.

Choice #2: Off-line Approach
→ DBMS monitors access logs offline and then makes decision about what data to move to DSM.

Choice #3: On-line Approach
→ DBMS tracks access patterns at runtime and then makes decision about what data to move to DSM.
PARTING THOUGHTS

A flexible architecture that supports a hybrid storage model is the next major trend in DBMSs.

This will enable relational DBMSs to support all known database workloads except for matrices in machine learning.
JOY’s DANK
TIPS FOR PROFILING
Consider a hot program $Z$ with two functions `foo` and `bar`.

How can we speed up $Z$ with only a debugger?
→ Randomly pause it during execution
→ Collect the function call stack
RANDOM PAUSE METHOD

Consider this scenario
→ Collected 10 call stack samples
→ Say 6 out of the 10 samples were in foo

What percentage of time was spent in foo?
→ Roughly 60% of the time was spent in foo
→ Accuracy increases with # of samples
AMDAHL’S LAW

Say we optimized foo to run 2 times faster
What’s the expected overall speedup?

→ \( p \) = percentage of time spent in optimized task
→ \( s \) = speed up for the optimized task
→ Overall speedup = \( \frac{1}{p} \cdot s \) = 1.4 times faster
AMDAHL’S LAW

Say we optimized **foo** to run 2 times faster
What’s the expected overall speedup ?
→ 60% of time spent in **foo** drops in half
→ 40% of time spent in **bar** unaffected

→ **p** = percentage of time spent in optimized task
→ **s** = speed up for the optimized task
→ Overall speedup = \[ \frac{1 + p}{1 + p} = 1.4 \text{ times faster} \]
AMDAHL’S LAW

\[
\frac{1}{0.6} + 2 \cdot \frac{1}{0.4} = 1.4 \text{ times faster}
\]

\[
\frac{1}{0.6} + 2 \cdot \frac{1}{0.4} = 1.4 \text{ times faster}
\]

Say we optimized \texttt{foo} to run 2 times faster
What’s the expected overall speedup?
→ 60% of time spent in \texttt{foo} drops in half
→ 40% of time spent in \texttt{bar} unaffected
PROFILING TOOLS FOR REAL

Choice #1: Valgrind
→ Heavyweight instrumentation framework with a lot of tools
→ Sophisticated visualization tools

Choice #2: Perf
→ Lightweight tool that can record different kinds of events
→ Console-oriented visualization tools
CHOICE #1: VALGRIND

Instrumentation framework for building dynamic analysis tools

→ **memcheck**: a memory error detector

→ **callgrind**: a call-graph generating profiler
CHOICE #1: VALGRIND

Instrumentation framework for building dynamic analysis tools
→ **memcheck**: a memory error detector
→ **callgrind**: a call-graph generating profiler

Using **callgrind** to profile the index test and Peloton in general:

```
$ valgrind --tool=callgrind --trace-children=yes ./tests/index_test

$ valgrind --tool=callgrind --trace-children=yes ./build/src/peloton -D data &> /dev/null
```
KCACHEGRIND

Profile data visualization tool

$ kcachegrind callgrind.out.12345
Cumulative Time Distribution
Cumulative Time Distribution

Callgraph View
CHOICE #2: PERF

Tool for using the performance counters subsystem in Linux.
→ -e = sample the event cycles at the user level only
→ -c = collect a sample every 2000 occurrences of event

$ perf record -e cycles:u -c 2000
./tests/index_test

Uses counters for tracking events
→ On counter overflow, the kernel records a sample
→ Sample contains info about program execution
PERF VISUALIZATION

We can also use `perf` to visualize the generated profile for our application.

$ perf report
We can also use `perf` to visualize the generated profile for our application.

```
$ perf report
```

<table>
<thead>
<tr>
<th>Samples</th>
<th>Event details</th>
<th>Function details</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.00%</td>
<td>index_test</td>
<td>ld-2.19.so do_lookup_x</td>
</tr>
<tr>
<td>25.00%</td>
<td>index_test</td>
<td>ld-2.19.so _dl_lookup_symbol_x</td>
</tr>
<tr>
<td>21.43%</td>
<td>index_test</td>
<td>ld-2.19.so _dl_relocate_object</td>
</tr>
<tr>
<td>7.14%</td>
<td>index_test</td>
<td>ld-2.19.so check_match.9458</td>
</tr>
<tr>
<td>3.57%</td>
<td>index_test</td>
<td>libstdc++.so.6.0.21 operator delete(void*)</td>
</tr>
<tr>
<td>1.79%</td>
<td>index_test</td>
<td>libstdc++.so.6.0.21 __dynamic_cast</td>
</tr>
<tr>
<td>1.79%</td>
<td>index_test</td>
<td>libstdc++.so.6.0.21 operator new(unsigned long)</td>
</tr>
<tr>
<td>1.79%</td>
<td>index_test</td>
<td>libpelotonpg.so.0.0.0 Json::Value::Value()</td>
</tr>
<tr>
<td>1.79%</td>
<td>index_test</td>
<td>libpelotonpg.so.0.0.0 peloton::Value::CompareWithoutNull(peloton::Value) const</td>
</tr>
<tr>
<td>1.79%</td>
<td>index_test</td>
<td>libc-2.19.so int_free</td>
</tr>
<tr>
<td>1.79%</td>
<td>index_test</td>
<td>libc-2.19.so __memcpy_sse2_unaligned</td>
</tr>
<tr>
<td>1.79%</td>
<td>index_test</td>
<td>libc-2.19.so _dl_addr</td>
</tr>
<tr>
<td>1.79%</td>
<td>index_test</td>
<td>libc-2.19.so __libc_dl_error_tsd</td>
</tr>
<tr>
<td>1.79%</td>
<td>index_test</td>
<td>ld-2.19.so strcmp</td>
</tr>
<tr>
<td>1.79%</td>
<td>index_test</td>
<td>index_test testing::TestEventListeners::TestEventListeners()</td>
</tr>
</tbody>
</table>
We can also use `perf` to visualize the generated profile for our application.

```
$ perf report
```

Cumulative Time Distribution
PERF EVENTS

Supports several other events like:
→ L1-dcache-load-misses
→ branch-misses

To see a list of events:

$ perf list

Another usage example:

$ perf record -e cycles,LLC-load-misses -c 2000 ./tests/index_test
REFERENCES

Valgrind
→ The Valgrind Quick Start Guide
→ Callgrind
→ Kcacheegrind
→ Tips for the Profiling/Optimization process

Perf
→ Perf Tutorial
→ Perf Examples
→ Perf Analysis Tools