The Homework Stream Database

A critical piece of infrastructure to support home network redesign within the Homework project is an in-memory stream database. This document describes the requirements that the database needs to meet, and discusses implementation details in order to meet those requirements.

Requirements

The primary research goal of the Homework Stream Database development within the Homework project is to exploit stream database techniques for storing measurement data about a home wireless network and supporting monitoring tasks that can drive policy-based management and network debugging. The database decouples the measurement tasks from the monitoring tasks, and provides stream SQL as the language for defining data schema and manipulating the stored data.

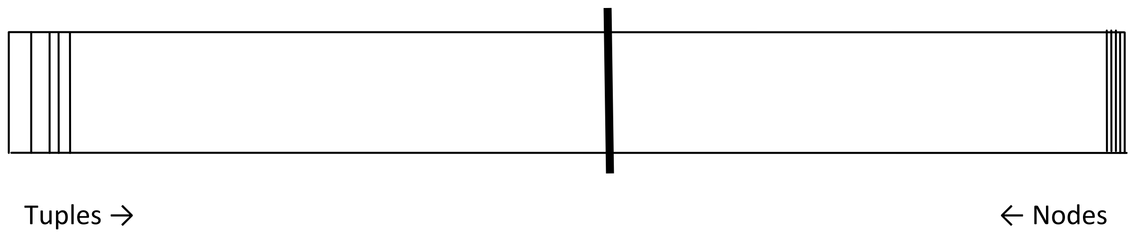
More requirements will be documented in the future regarding required capacity of the database, minimal response times for insert and select queries, and other performance aspects.

Implementation Details

The database implements a stream database []. The scope of queries for most stream databases is the set of tuples that pass an observation point during a window in time. For the Homework stream database, a fixed-size memory buffer is used to store tuples as they arrive, with appropriate control structures to link these tuples into tables. The memory buffer is treated like a circular buffer; when the write pointer for the buffer overtakes the first used pointer, the oldest tuple[s] are removed from the buffer to make room for new tuples. In this way, the scope over which queries are executed changes over time.

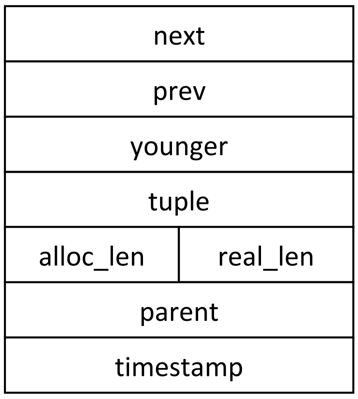
The base level infrastructure for tuples and nodes

A large, statically allocated, array of unsigned characters is declared to hold data tuples and metadata nodes that are required to manage the data.



On the first pass through the buffer, tuples are allocated moving toward high memory and nodes are allocated moving toward low memory; when the two allocations meet, the number of nodes that can be used for tuples is fixed, and the lower portion is treated as a circular buffer, with new tuples overwriting the oldest tuples. The size of the buffer is a defined constant, MB_SIZE.

The nodes have the following structure:



where *next* and *prev* are used to maintain a doubly-linked list of nodes in a table, *younger* is used to maintain a singly-linked list of all tuples in the database by insertion time (enabling overwriting of the oldest tuple given the circular buffer semantics), *tuple* is a pointer to the location in the memory buffer in which the tuple is stored, *alloc_len* is the amount of space allocated in the buffer for the tuple, *real_len* is the actual length as supplied by the user, *parent* is a pointer to the table in which this tuple is stored, and *timestamp* is the time at which the tuple was added to the buffer.

General dynamic memory support

Besides storage for tuples data and metadata, supported by the previous infrastructure, there are a number of additional data structures that must be allocated dynamically while the system is operational. For example, the current system uses a hash table to map from a table name to a Table data structure (more on these structures in the next section). When parsing an SQL query, temporary parse tree nodes must be created. One must be able to store a dynamic number of published queries against a table. These, and many other structures, cry out for malloc() and free().

While there are kmalloc() and kfree() defined in the Linux kernel, this dynamic storage is used by all kernel modules for their dynamic allocation requirements. There is no way to bound the amount of storage that is used by the database, with the potential of causing the system to freeze or become sluggish due to overuse of the kmalloc resources. For the in-kernel version of the database, we will supply a simple, first fit dynamic storage module with malloc, free, and calloc entry points identical to those in the standard C library (via a set of #define's). A defined constant HEAP_SIZE determines the amount of memory that is available for use in this dynamic storage module.

Table support

Above this base level infrastructure for tuples and node metadata, the database organizes tuples into tables. Each table has a defined schema (names and types for each row in the database); this schema is established when the table is dynamically created.

Each table has a corresponding Table structure allocated from the heap, and it is stored away in a hash table, where the table name is used to generate the hash key.

A Table has the following structure:

ncols
colname
coltype
oldest
newest
count
sublist

where *ncols* is the number of columns defined for tuples in this table, *colname* is an array of char * pointers to the names of the columns, *coltype* is an array of int * pointers to the types of the columns, *oldest* and *newest* are used to maintain a doubly-linked list of nodes in the table, *count* is the number of nodes currently in the table, and *sublist* is a linked list of subscriptions to the table.

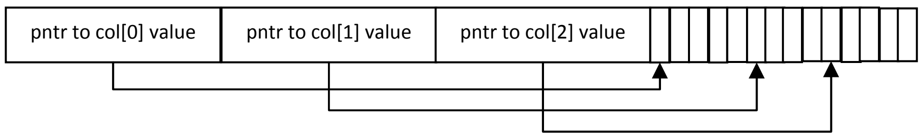
Tuple representation in the memory buffer

Each tuple, however it is represented for SQL processing, is stored as an array of bytes in the memory buffer. Due to potential alignment issues of multi-byte data types, the actual storage for each byte array is forced to be 4-byte or 8-byte aligned, depending upon whether we are operating on a 32/64 bit machine. The actual computation for the amount of storage required for a particular tuple of length N is

```
length = ((N – 1) / ALIGNMENT + 1) * ALIGNMENT
```

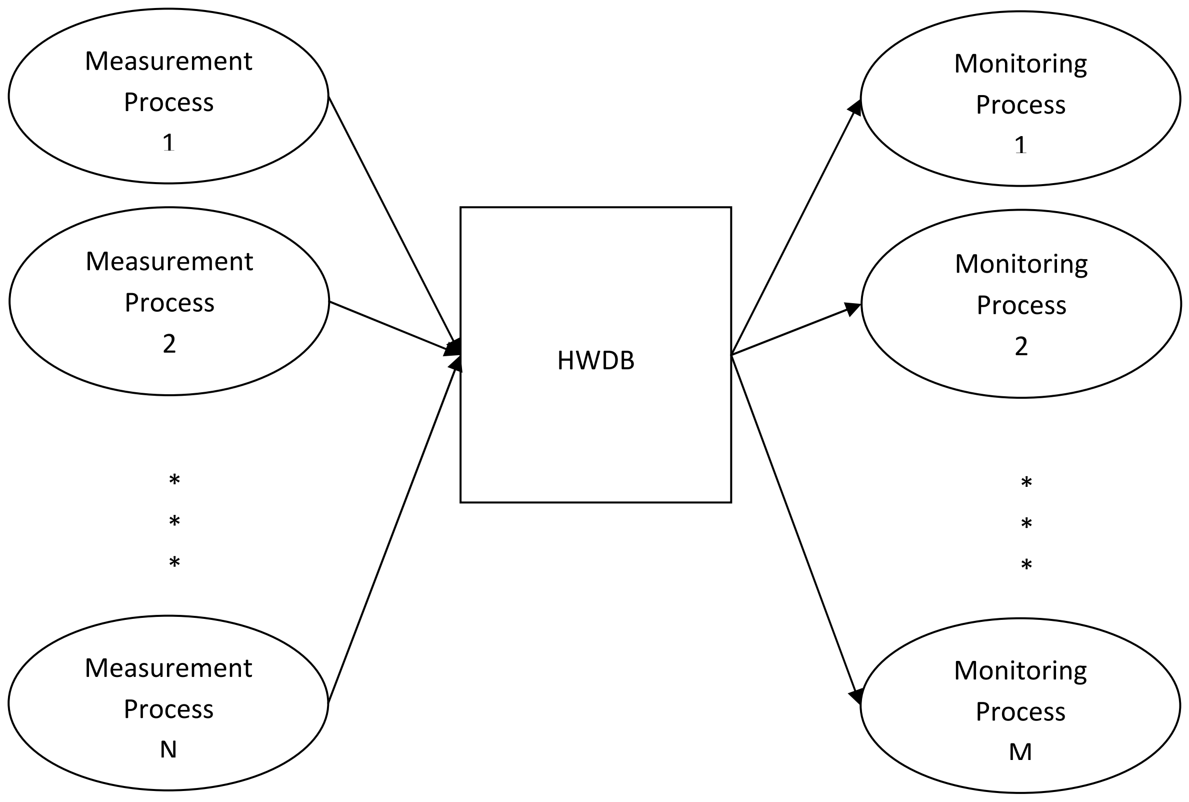
where `ALIGNMENT` has the value of 4 on a 32-bit architecture and 8 on a 64-bit architecture.

Each tuple consists of values for each of the defined columns for its enclosing table. The current implementation stores all values as 0-byte terminated strings. The actual layout of these values within the allocated byte array in the circular buffer is as follows (assuming a table with 3 columns):



Measurement and monitoring architecture

The following figure shows the measurement and monitoring architecture in which the database is positioned.



Each of the measurement processes "insert"s tuples into an appropriate table in the database at times appropriate to whatever the process is measuring. Each of the monitoring processes issues "select" queries to tables at times appropriate to its monitoring task. At the current time, all of the "select" queries should be pull queries from the monitoring processes, as the publish/subscribe facilities in HWDB do not scale with rapid additions to the database. Future versions of the database will enable appropriate subscription of queries to tables in the database, such that the communication from HWDB to monitoring processes can be push queries.

As an example of a measurement process, the "sniff" application uses the pcap library to capture all packets on the wireless interface of the homework router. For each packet seen, an in-memory entry for its corresponding flow (protocol, src IP, dst IP, src port, dst port) is updated to note the arrival of another packet and the additional bytes "on the wire" of that packet. A separate thread in "sniff" wakes up once a second, collects all of the in-memory flow entries, zeroes those entries, and then generates and invokes a BULK insert rpc to the database for the collected flow entries.

The "flowmonitor" monitoring process is discussed in the design note entitled "Periodic HWDB Client Architecture".