**INSTALLATION**

Extract .zip and add as a project in Eclipse. Run the server startup as usual. Performance tests can be run from simpledb.datagenerator. Otherwise, the default studentClient database functions just as well.

**TESTING QUERIES**

The test queries from the previous project (pokemonTest.simpledb.\*, studentClient.simpledb.\*) as well as the performance testing queries were used in order to test whether the database still functioned properly.

**DESIGN**

Index

The Index data structure was updated throughout simpleDB in order to accommodate the index type based on user input; “bt” and “eh” will create B-tree and extensible hash indices respectively, and any other input (or lack thereof) will make a static hash index.

Extensible Hashing

The Extensible Hash index (simpledb.index.exhash) expands on the functionality of the original hash index class. Initially, 4 buckets are created and the global depth is set to 2 (2-bit checks). Buckets are now a new class rather than an abstract integer and hold an ID number, local depth, contents, and capacity. The contents can not be larger than 4. Buckets are held in a hashmap to match ID numbers to buckets during hashing.

A bitmask is used to determine the bucket to send a record to (2 => b11, 3 => b111, etc). For each record, the bitmask is initalized based on the global depth for safety and a bucket number is generated with the search key's hash and the bitmask. If the local depth of this bucket is not equal to the global depth, the bitmask and bucket number are updated before continuing. This is how our design handles multiple local depths; only the necessary number of bits are checked.

If a bucket is not full, the record is added to its contents and the index is updated. Otherwise, the local depth (and the global depth if need be) are incremented. The hashed, unmasked value of the record, as well as this bucket, are then passed to a bucket updater for splitting and proper hashing.

The bucket updater first applies the bitmask to the hashed value, then uses this to make a new bucket. The contents of the old bucket are removed and rehashed to update these two buckets, and finally, the record that caused this mess is put into its proper bucket and the index is updated.

SmartMergeJoin

ExploitSortQueryPlanner

**TESTING**

Performance tests were run with 5,000 queries per table (anything higher did not complete after several hours, despite running on a very powerful machine).

Extensible Hashing

SmartMergeJoin

**BUGS**

We are unaware of any bugs in this application. However, we are not sure if our IO counter for select statements works as intended.