Checkpoint3

- **Overview**: Add a pre-processing phase to your system.
- **Deadline**: May 8
- **Grade**: 15% of Project Component
  - 5% Correctness
  - 5% Efficiency
  - 5% Code Review

Once again, we will be tightening performance constraints. You will be expected to complete queries in seconds, rather than tens of seconds as before. This time however, you will be given a few minutes alone with the data before we start timing you.

Concretely, you will be given a period of up to 5 minutes that we'll call the Load Phase. During the load phase, you will have access to the data, as well as a database directory that will not be erased in between runs of your application. Example uses for this time include building indexes or gathering statistics about the data for use in cost-based estimation.

Additionally, CREATE TABLE statements are now annotated with PRIMARY KEY and FOREIGN KEY attributes. You may hardcode index selections for the TPC-H benchmark based on your own experimentation.

BerkeleyDB

For this project, you will get access to a new library: BerkeleyDB (Java Edition). Don't let the name mislead you, BDB is not actually a full database system. Rather, BDB implements the indexing and persistence layers of a database system. Download BDB at:

http://odin.cse.buffalo.edu/resources/berkeleydb/berkeleydb.jar

The BerkeleyDB documentation is mirrored at:

http://odin.cse.buffalo.edu/resources/berkeleydb/

You can find a getting started guide at:

http://odin.cse.buffalo.edu/resources/berkeleydb/GettingStartedGuide

And the javadoc at:

http://odin.cse.buffalo.edu/resources/berkeleydb/java/

BDB can be used in two ways: The Direct Persistence layer, and the Base API. The Direct Persistence Layer is easier to use at first, as it handles index management and serialization through compiler annotations. However, this ease comes at the cost of flexibility. Especially if you plan to use secondary indexes, you may find it substantially easier to work with the Base API. For this reason, this summary will focus on the Base API.

Environments and Databases

A relation or table is represented in BDB as a Database, which is grouped into units of storage called an Environment. The first thing that you should to do in the pre-computation phase is to create an Environment and one or more Databases. **Be absolutely sure toclose both the environment and the database before you exit**, as not doing so could lead to file corruption.

BDB Databases are in effect clustered indexes, which means that every record stored in one is identified (and sorted by) a key. A database supports efficient access to records or ranges of records based on their keys.

Representing, Storing, and Reading Tuples

Every tuple must be marked with a primary key, and may include one or more secondary keys. In the Base API, both the value and its key are represented as a string of bytes. Both key and value must bestored as a byte array encapsulated in a DatabaseEntry object. Secondary Keys are defined when creating a secondary index.

Note that you will need to manually extract the key from the rest of the record and write some code to serialize the record and the key into byte arrays. You could use toString(), but you may find it substantially faster to use Java's native object serialization:

ObjectOutputStream | ObjectInputStream

… or a pair of classes that java provides for serializing primitive data:

DataOutputStream | DataInputStream

Like a Hash-Map, BDB supports a simple get/put interface. Tuples can be stored or looked up by their key. Like your code, BDB also provides an iterator interface called a Cursor. Of note, BDB's cursor interface supports index lookups.

Secondary Indexes

The Database represents a clustered index. In addition, BDB has support for unclustered indexes, which it calls SecondaryDatabases. As an unclustered index, a secondary database doesn't dictate how the tuples themselves are laid out, but still allows for (mostly) efficient lookups for secondary "keys". The term "keys" is in quotation marks, because unlike the primary key used in the primary database, a secondary database allows for multiple records with the same secondary key.
To automate the management process, a secondary index is defined using an implementation of SecondaryKeyCreator. This class should map record DatabaseEntry objects to a (not necessarily unique) DatabaseEntry object that acts as a secondary key.

BDB Joins

Another misnomer, BDB allows you to define so-called Join Cursors. This is **not** a relational join in the traditional sense. Rather, a Join Cursor allows you to define multiple **equality** predicates over the base relation and scan over all records that match all of the specified lookup conditions.

Performance Tuning

BerkeleyDB can be quite tricky to get performance out of. There are a number of options, and ways of interacting with it that can help you get the most out of this indexing software. Since evaluation on the grading boxes takes time due to the end-to-end testing process, I encourage you to evaluate on your own machines. For best results, be sure to store your database on an HDD (Results from SSDs will not be representative of the grading boxes). Recall that the grader boxes have 4 GB of RAM.

Heap Scans

Depending on how you've implemented deserialization of the raw data files, you may find it faster to read directly from the clustered index rather than from the data file. In the reference implementation, reading from a clustered index is about twice as fast as from a data file, but this performance boost stems from several factors. If you choose to do this, take a look at DiskOrderedCursor, which my experiments show is roughly about twice as fast as a regular in-order Cursor on an HDD on a fully compacted relation.

Locking Policies

Locking is slow. Consistency is slow. As long as you're not implementing your code multithreaded or with updates or transactions, you'll find that cursor operations will be faster under LockMode.READ_UNCOMMITTED. See below for ways to set this parameter globally.

Config Options

BDB also has numerous options that will affect the performance of your system. Several options you may wish to evaluate, both for the load and run phases:

- EnvironmentConfig
  - setCachePercent
  - setLocking
  - setConfigParam
    - EnvironmentConfig.ENV_RUN_CLEANER
    - EnvironmentConfig.ENV_RUN_CHECKPOINTER
      - See the documentation for Environment.cleanLog() if you plan to turn either of these off.
- DatabaseConfig and SecondaryConfig
  - setReadOnly
  - setTransactional
- DiskOrderedCursorConfig
  - setInternalMemoryLimit

- setQueueSize

Interface

Your code will be evaluated in exactly the same way as Projects 1 and 2. Your code will be presented with a 500MB (SF 0.5) TPC-H dataset. Before grading begins, your code will be run once to preprocess the data. You will have up to 5 minutes, after which your process will be killed (if it has not yet terminated). Your code will then be run on the test suite.

As before, your code will be invoked with the data directory and the relevant SQL files. Two additional parameters will be used in the preprocessing stage:

- `--db directory`: A directory in which it is safe to persist data. The contents of this directory will be persisted across the entire grading run.
- `--load`: This parameter will be passed during the preprocessing phase. When it appears on the command line, you have up to 5 minutes to preprocess the data.

```
java -cp build:jsqlparser.jar:...

     edu.buffalo.cse562.Main

     --data [data]

     --db   [db]

     --load

     [sqlfile1] [sqlfile2] ...
```

This example uses the following directories and files:

- `[data]`: Table data stored in '|' separated files. As before, table names match the names provided in the matching CREATE TABLE with the .dat suffix.
- `[db]`: A directory for permanent data files. This directory will be persisted across all runs of the

- `[sqlfileX]`: A file containing CREATE TABLE and SELECT statements, defining the schema of the dataset and the query to process. If –load appears on the command line, these files will contain only CREATE TABLE statements.

Grading

Your code will be subjected to a sequence of test cases and evaluated on speed and correctness.

- **0/10 (F)**: Your submission does not compile, does not produce correct output, or fails in some other way. Resubmission is highly encouraged.
- **5/10 (C)**: Your submission runs the test query faster than the C threshold (listed below for each query), and produces the correct output.
- **7.5/10 (B)**: Your submission runs the test query faster than the B threshold (listed below for each query), and produces the correct output.
- **10/10 (A)**: Your submission runs the test query faster than the A threshold (listed below for each query), and produces the correct output.

TPC-H QUERY	GRADE	MAXIMUM RUN-TIME (S)
Q1	A	30
	B	60
	C	90
Q3	A	5
	B	30
	C	120
Q5	A	10
	B	60

	C	120
Q6	**A**	20
	B	45
	C	70
Q10	**A**	10
	B	30
	C	90
Q12	**A**	40
	B	60
	C	90