Slingshot: A modular framework for designing data processing systems

Bogdan Simion, Daniel N. Ilha, Suprio Ray, Leslie Barron, Angela Demke Brown, Ryan Johnson
Department of Computer Science, University of Toronto
{bogdan, suprio, demke, ryan.johnson} @cs.toronto.edu,
daniel.n.ilha@gmail.com, leslie.barron@utoronto.ca

Abstract

Traditional relational database engines have been losing ground to specialized data processing engines in virtually every market segment, from data warehousing, OLTP, and stream processing, to scientific applications [1], [2], [3]. Although relational database engines are evolving to leverage new technologies and more efficient processing paradigms, the generality of a large monolithic engine often makes this a significant effort.

Our aim is to delimit database engine components and decouple their functionality to design a more lightweight and flexible data processing engine that can support any application domain efficiently and without the effort of a complete redesign. We introduce a new data processing engine called Slingshot, where modularity and implementation flexibility are the top priority. In Slingshot, the core database engine is minimal and mainly handles inter-operation of the database components. Each component, abstracted by an interface, can be externally implemented and plugged into the framework as a module that handles the component's functionality. As a result, component design decisions are offloaded to the modules, which allows designers the liberty to choose what features are suitable for their target applications, to drop excess functionality, and to optimize code independent of the rest of the engine.

We compare Slingshot to a traditional RDBMS and to custom solutions on queries that are representative of three application types (spatial, OLAP, and OLTP). We show that Slingshot outperforms the RDBMS in most cases, while performing comparably in others. Furthermoe, Slingshot performs better or comparable to custom solutions on most tests.

Finally, we show that Slingshot's flexibility allows for easy and efficient integration of GPU support for spatial refinement, resulting in speedups of up to 70x over traditional database engines on long-running spatial analytics queries.

I. Introduction

Relational database management systems (RDBMS) have a respectable pedigree, and are being used for a wide variety of applications, including business data processing (OLTP), data warehousing and analytics (OLAP), stream processing, scientific data processing, and others. RDBMSs include many optimizations, UDFs, an expressive query language, etc., however, they were designed in an era when the architectural landscape and computational demands of applications were much different.

Big database manufacturers are making efforts to support the deluge of architectural advances (e.g., vector instructions, larger caches, multicores, GPUs, FPGAs, or fast SSDs and PCM) but adding new features or adapting components in their existing codebases is a highly difficult task. RDBMS codebases tend to be packed with legacy features and functionality needed to support a wide range of applications and workload types with varying demands. The resulting RDBMS codebases are often huge, complex, and hard to modify or adapt.

These considerations have caused many researchers to roll out custom solutions dedicated for their target application or workload. Examples include column-stores (such as C-Store [4] and its spinoff Vertica, MonetDB [5], or ParAccel [6]) dedicated for data warehousing applications, and a plethora of NoSQL solutions and key-value stores.

Stonebraker et al. [1], [2] observe that major RDBMSs are lagging behind specialized engines in the data warehouse, stream processing, and scientific database markets. In some cases, custom solutions outperform traditional RDBMSs by 1-2 orders of magnitude. Stonebraker et al. [3] showed that their custom prototype, H-Store, can beat current RDBMS engines by nearly 2 orders of magnitude on the OLTP market segment as well (one of the remaining key selling points for RDBMS vendors). For spatial database workloads, custom solutions also outperform traditional RDBMSs. We showed [7] that PostgreSQL/PostGIS is several orders of magnitude slower than a custom R-tree implementation, due to the pitfalls of its general indexing framework.

The new custom solutions are designed to perform well (in terms of latency or throughput), yet they are only applicable to a narrow range of applications. On the other end of the spectrum, all-purpose RDBMSs have (i) limited flexibility, making it difficult to integrate new features efficiently, and (ii) feature bloating - supporting everything hurts performance compared to lightweight single-purpose solutions.

Our goal is to design a lightweight and modular data processing framework that allows flexibility and efficiency. Although our immediate target is to provide improvements for spatial indexing, we do not limit our endeavour to simply provide an efficient indexing framework, but rather explore a general design at the level of the entire database engine. The framework design should allow database designers to extend or replace components (e.g., the indexing component, storage manager, transaction manager, etc.) with new ones that support new features and functionality, or which are better optimized for the ever-changing architectural landscape. With modularity as a paramount goal, our design should permit new components to be plugged into the processing framework independently of other components.

The traditional complex, overweight RDBMS bears resemblance to a monolithic operating system (OS). The components of the database engine are tightly coupled and dropping functionality or customizing the internals is difficult (although some attempts exist, as we shall discuss in Section II). Our modular design is inspired by the microkernel OS concept, in which the kernel is stripped down to minimal functionality while traditional OS services, such as file systems and device drivers, are run as separate server processes in user space. Server processes communicate via facilities provided by the microkernel and can be replaced with different implementations independently. The result is a more robust, flexible, and customizable OS, which can achieve higher performance for workloads that benefit from specialized services. Unfortunately, while the microkernel idea has many software engineering benefits, the cost of reflecting all communication between user-level system servers through the microkernel proved to be significant. In the database domain however, we are not faced with the same protection boundary penalty, so achieving both modularity/customizability and high performance is feasible.

We propose Slingshot, a lightweight data processing framework, which aims to decouple database components into external modules that can be implemented efficiently by database designers, independently of all other modules. Slingshot delimits key components of the database based on their role, and exposes them as interfaces for external implementation. Most design decisions and optimizations for each component are delegated to external modules. To our knowledge, Slingshot is the first data processing framework designed to act similarly to a microkernel, and target both modularity and performance. We show that generality need not sacrifice performance, if abstractions are designed carefully with flexibility in mind.

In Slingshot, adapting the database engine to suit various application demands is much easier than in monolithic DBMS engines. By simply swapping one or more modules serving the right functionalities, the framework can be tailored to a new workload. For example, to better serve an OLAP workload, we can design and load a new data layout module, which implements column-store functionality, and a new storage manager module optimized for read-mostly non-transactional requests. Similarly, for applications that need transactional semantics but can drop certain properties (e.g., durability), the corresponding module can be designed accordingly (e.g., by relaxing the functionality in the storage manager that handles data persistence and dropping the logging module altogether).

By delimiting boundaries between the various components using clearly defined interfaces, Slingshot provides:

i) flexibility in choosing the appropriate modules for the target data processing application (e.g., OLTP workloads have different demands than OLAP workloads).

ii) freedom to implement functionality as efficiently as possible in external modules (e.g., using optimized data structures or code that exploits hardware features such as AVX extensions) without worrying about cross-module dependencies.

iii) the ability to "shed" functionality for performance, by allowing module implementations to omit unneeded features.

In a nutshell, Slingshot is modular, efficient and lightweight.

Our goal is not to plug in third-party databases, query engines, or storage managers, that were not designed to be modular (or are limited in the level of modularity exposed). Our goal is to offer an interface that allows writing database components as modules into a lightweight data processing engine. The benefit of isolating modules to avoid cross-component code dependencies is not necessarily a reduction in lines of code, but rather in the fact that the development effort does not involve going through hundreds of thousands of lines of existing code to patch in new functionality (that might end up performing poorly anyway if that codebase was not designed to efficiently support the new functionality to begin with).

II. Drawing lines of abstraction

Determining *which* abstraction lines can be drawn and *where* is not straightforward. Cross-cutting concerns are aspects of a database engine that permeate into multiple logical components; they introduce dependencies that make the database engine cluttered, hard to extend, and possibly inefficient. We argue that the dependencies can be lifted by designing the database engine in a more modular fashion. Some existing database engines have attempted to generalize certain components or provide customizability. We draw from their past experiences while being aware of their limitations.

In Oracle's BerkeleyDB [8], users can choose some build options that suit their needs. The ability to strip out some features for performance is important and we incorporate this concept in Slingshot's modular design. Unfortunately, BerkeleyDB has no good way of plugging in new implementations.

At the storage level, MySQL [9] supports several storage engines (e.g., InnoDB, IBMDB2I, MyISAM, MRG_MyISAM, Memory, etc.) A new Storage Manager can be introduced by implementing a *table handler* for it, which MySQL uses to distinguish between different storage engines. The Storage Manager takes care of all aspects related to storage: data layout, tables, row operations, indexes, transactions - in conjunction with the Transaction Manager and Lock Manager, which are not clearly defined or separated. Although MySQL exposes an interface to implement a custom Storage Manager, in reality, a designer has to understand a large portion of the MySQL engine internals and data structures.

For example, the index code is intertwined with the Storage Manager, which makes it difficult to implement new indexes

without gaining a deep understanding of a large portion of the code. [1] Such cross-cutting concerns between logical components make it challenging to introduce new extensions to a large, pre-existing codebase. To illustrate the magnitude of the task of understanding the MySQL codebase, we note that the InnoDB storage engine has 16K lines of code (LOC) just in the handler, which mainly includes basic stubs that offload the functionality to other helper functions spread into numerous other files. The entire MySQL codebase is 1.53 Million LOC.

Slingshot extends MySQL's idea of replaceable storage engines by allowing *any component* to be custom-designed and easily pluggable. However, in Slingshot the cross-cutting concerns are handled by the core framework, not by implementer-defined modules. This decoupling makes it easier for the implementer to create a module without delving deep into the database engine internals. It also simplifies the task of implementing new features, like new data layouts, new indexes, etc., by eliminating the need to worry about the logical dependencies among components.

PostgreSQL [10] also has a monolithic design but like many other RDBMSs it allows extensibility through user-defined functions (UDFs). The UDF integration with the query optimizer is a positive aspect, but in practice their pluggable access methods may fall short. For example, the Generalized index Search Tree (GiST) framework provides a set of methods that can be used to implement new index types. The PostGIS spatial extension implements these methods to provide an Rtree index, however the GiST-based Rtree can be outperformed by a custom implementation by orders of magnitude [7]. We argue the GiST draws the abstraction line in the wrong place, limiting the implementation options for the index designer and causing performance degradation. A generalized indexing framework should specify high-level index operations (e.g., create, search, update, etc.), rather than how these operations should be performed (e.g., GiST's PickSplit, Union, etc.). Slingshot aims to provide modularity while allowing components to be designed and optimized independently so that good performance is also achieved.

Given that database engine components have strong interconnections, it may not be enough to use a specific component design within a different engine without considering the ramifications. For example, Abadi et al. [11] argue that one cannot obtain the same performance as a column-store by vertically partitioning the schema of a row-store and indexing every column; they argue that both the storage manager and the query executor must be adapted to achieve some of the performance benefits of a column-store. We considered this work when designing Slingshot, in order to define the interfaces for database components and the microkernel-style core Slingshot framework carefully, so that dependencies would not hinder either performance or customizing functionality independently.

In the next sections, we describe the design of the Slingshot framework, the interaction between components, and how new modules can be implemented and plugged into the framework. We then revisit how this design helps alleviate cross-cutting concerns in a database engine. We evaluate Slingshot on three types of applications: a set of common spatial queries, a workload composed of a subset of OLAP-type queries, and a set of queries from a standard OLTP benchmark. Although Slingshot does not claim to have the most efficient implementation in the modules that handle the functionality of database components, we show that it can outperform traditional RDBMSs in most cases. We also show that Slingshot approaches the performance of custom solutions on their target workload, due to a lightweight core engine and independently-designed modules.

III. Design

Slingshot is a modular data processing *framework*, which aims to provide functionality through a core (immutable and minimal) infrastructure and a set of well-defined public interfaces for externally-pluggable modules. The goal is to abstract database *components* (e.g., indexing, storage, data manipulation, etc.), in a way that separates the inter-component dependencies from the external modules. Specialized modules for different application types can then be developed independently and easily replaced without any framework changes.

Each *interface* defines the role and attributes of the corresponding component, while an external module implements the interface. For each component, the *external implementation* or *module* is left up to the implementer and can be loaded dynamically as long as it abides by the corresponding interface conventions. To allow component inter-operability, an *internal implementation* for each component is included in the Slingshot framework. The *internal implementation* of the interface performs a minimum amount of work (as needed) to allow for some internal bookkeeping and to facilitate inter-operation with other components. All other functionality is offloaded to the external implementation. As a result, the internal implementation also calls into the external module (a dynamic shared object) that is registered to handle the respective interface functions.

In designing Slingshot, we used a set of ad-hoc *guiding principles* based on experiences with other systems and our own experience delving into various database engine internals.

1. **Distinguish**. When designing an interface, only expose what operations the designers should implement, not how they should implement it. Although this principle sounds simple, database engines do not always follow this guideline.

[1] e.g., one would have to figure out that the predefined TABLE struct contains a key_info member that contains information about the table's index type and behaviour, and then go change a whole set of data structures and functions just to indicate a new index is present, as well as understand the dependencies between table and index methods.

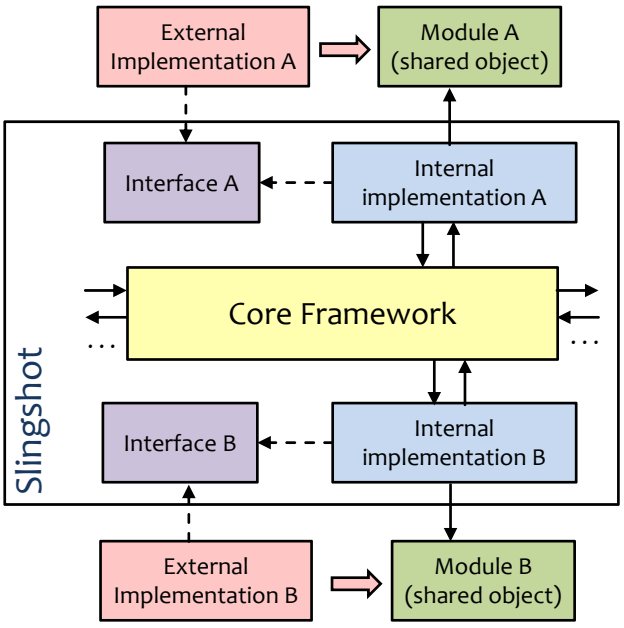

Fig. 1. Slingshot Components - Interfaces, Implementation and Interoperation

2. **Decouple**. Each method exposed through an interface A should not make any implementation assumptions about the existence of another component B or how A and B inter-operate. Any functionality that involves a cross-cutting dependency between the two components is the responsibility of the core framework, which handles interconnectivity. For example, a module implementing a Table data layout should make no assumption about the possibility of indexes being available on the columns touched. As a result, implementing a Table interface method that inserts records should be strictly about adding the data to the correct location. The update of any possibly-existing indexes is handled by the core framework which, i) knows which indexes correspond to the affected columns, and ii) knows which method to invoke from which Index module to handle the index insertion.

3. **Delineate**. The last principle goes hand in hand with the second one, and concerns drawing borders around each database component in such a way that its functionality can be handled strictly independently of all other components. This principle is more in the sense of logical functionality: try to separate logical components as fine-grained as needed, such that the logical functionality to be implemented in the module does not compel or simply mislead the designer to break decoupling or to bundle extra functionality that does not belong in that module.

Following these guiding principles, Slingshot captures the main components of a data management system and defines the functionality for each of them through clear and minimal *interfaces*. The interfaces defined by Slingshot are all double-sided abstraction layers through which a module can provide its functionality and also interact with the other modules. The framework takes care of internal tracking of tables, indexes, and generally minimal operations that are necessary for all these components to inter-operate.

The interface to each component exposes to a designer the operations needed to implement a module that handles the functionality of the component. The interface is as generic as necessary to allow the designer the liberty of optimizing external modules for their target application (e.g., implementing a column store in the Data Layout component, optimizing the Storage Manager for performance by relaxing some transactional constraints, etc.).

The *external modules* that implement the interfaces exposed by the Slingshot framework are implemented using *loadable shared-object libraries* (or DLLs in Windows, although currently we have not yet ported our framework to other platforms). This allows for flexibility in choosing which modules we wish to use for a particular type of application, without changing the Slingshot framework internals. In contrast, most available RDBMSs are monolithic and lack the desired flexibility (for example, configuring an RDBMS to switch data layouts or shed features that may cause performance overhead). We implement modules for each critical component in Slingshot, to evaluate the performance of our framework on various application types (Section VII).

In Figure 1, we show an example of how two components A and B are implemented in Slingshot. The interface for component A is exposed to the external implementer. The external implementation of A must implement (represented using a dashed arrow) interface A and compiles (wide arrow) into a shared object - the Module A. The internal implementation of A implements the interface as well, but only performs a minimal amount of work to allow interoperability with other components. To do so, it can

perform function calls (represented using a regular solid arrow) into other components' internal implementation through the core framework. Additionally, the internal implementation performs component A's functionality by calling into the external module. Component B follows the same idea, as do all other components (represented using '...').

A. Framework Core

At the *core level*, the Slingshot framework provides: a *type abstraction layer*, through which external modules can register new types and functions that can then be used to manipulate them; and a *module management* system that offers the means to load and unload modules dynamically into the framework, and keeps track of currently loaded modules. These core components cannot be replaced at run time - they are essential to the interface structure and must be available and consistent at all times. Additionally, there is only a small variety of changes that could be made to these two components, which does not justify the additional cost of abstracting them as well. The type abstraction layer is not as interesting for the purposes of this work, so we omit the details.

The Module Manager takes care of loading modules and maintaining information about the loaded modules, so that the Executor can decide which modules to use or to ask for a given functionality. In *debug mode*, the Module Manager also keeps track of the resources used by any particular module to help developers debug bottlenecks and similar issues. Note that it can only keep track of resources allocated through the Slingshot-defined methods; any resources allocated directly by the modules themselves are not visible to the Module Manager.

In Slingshot, any dynamic library containing code that implements an interface is called a *package*. Before the package can be loaded, a module implementer must define a *package descriptor*, which contains metadata about the name and location of the shared library, as well as the module(s) it contains. For example, a package which implements a spatial index module into Slingshot may have the following package descriptor "SpatialIndex.pak":

```
SpatialIndex
modules/bin/rtree.so
RTreeIndex
```

The first line contains the name of the package, the second entry is the location of the dynamic library containing the module, and the last line of the file contains the module name.

In Slingshot, all entities (such as tables, indexes, or even columns) are managed internally as objects. Each table or index object is identified by a unique 64-bit ObjectID (OID) and is backed by one (or more) file(s) containing its corresponding data. For column-stores, we can also view columns as separate objects with individual OIDs, each stored in a separate file and linked to their parent table, as described later. This functionality is also immutable, so the *object management system* is part of the core framework. We refer to the ensemble of the object management system and the type abstraction layer of the core framework as the *Data Type System*.

IV. Slingshot Architecture

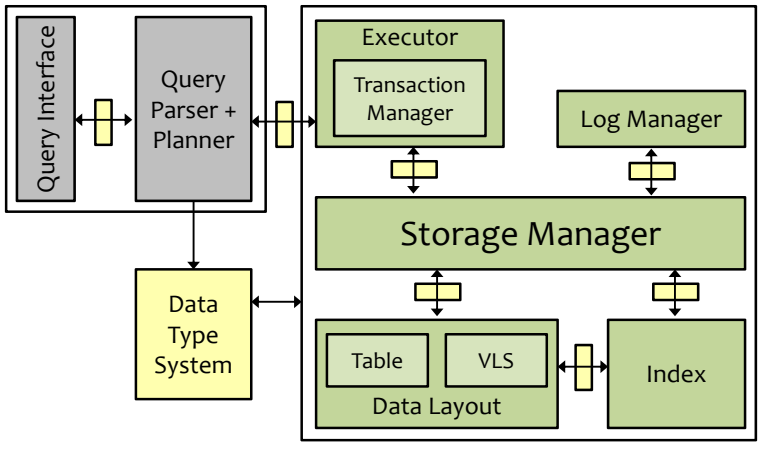

Fig. 2. Slingshot architecture

Figure 2 shows the components of the Slingshot framework and the most important connections between them. The small boxes over the arrows indicate calls through the Slingshot framework, to handle component inter-operation. The Query Interface is the client-end which receives user queries or commands, and passes them on to the Executor. The QueryParser and QueryPlanner components take care of generating an optimal execution plan, while the Executor handles the execution of the plan and is strongly connected with the Transaction Manager. The Transaction Manager takes care of enqueuing transactions

(or single queries, in case we do not want to use a transactional system) for processing and takes care of any conflict detection and resolution mechanisms. The Log Manager logs the transactions in flight, for fault tolerance and recovery purposes. All requests for data (disk-based or in-memory) by in-flight queries, as well as caching policy are handled by the Storage Manager. The Data Layout module defines how data is physically laid out on disk. Its Table subcomponent defines how to store and retrieve fixed-size records, while the Variable-Length Storage (VLS) module defines how variable-size data is managed. Various strategies for indexing data are included in the Index component.

We describe the details of these components in the following subsections, with the exception of the front end components (the query interface, parser and optimizer). We plan to design and implement these front end components in the future, but they are not essential to demonstrate the benefits of the modular database design. Furthermore, based on profiling several DBMS engines, the query parser and optimizer account for a negligible fraction of the entire query execution, hence they are not likely to impact our performance results either. We do however briefly discuss how the *Query Optimizer* interacts with other components, by detailing what statistics are collected to issue an efficient query plan, and which components are responsible for each piece of metadata.

A. DataLayout Component

The DataLayout component is responsible for the data layout on disk and the operations required to manipulate the data (create, retrieve, insert, update, remove, etc.). It is composed of two sub-interfaces: the Table interface and the Variable-Length Storage (VLS) interface. The internal side of the DataLayout component also keeps track of the tables registered (based on their OID) by interacting internally with the Object Management system. For example, when a Table is created, the internal implementation of the DataLayout component invokes the Object Management system to register the new table and obtain a new unique OID.

1) I. Table: The Table component defines the interface for storing fixed-size records (e.g., int, long, float, fixed-size char array, date, etc.). In keeping with our guiding design principles, the interface defines its role and attributes through a generic set of methods that handle table operations (creating a table, retrieving the number of columns, column types, adding or removing columns, inserting/updating/removing/retrieving rows, etc.) and an extensible data structure for table metadata. The external implementation module has to abide by the interface by implementing these methods and extending the Table structure, but is allowed flexibility in the choice of data layout and optimizations for record management operations.

A. Interface The interface defines 3 elements. a) The *table entity* is described in a Table structure containing minimal table metadata (e.g., OID, number of rows, etc.), as well as a pointer to an externally loaded table module containing the table "functionality" (see Listing 1).

Listing 1. Table structure.

```
struct Table {
   ObjectID oid;
   TableModule * dlm; /*dynamically loaded module*/
   size_t num_rows;
};
```

b) *External interface*: The TableModule interface contains a set of methods that handle table operations. These include creating a table, retrieving the number of columns, column types, adding or removing columns, inserting/updating/removing/retrieving rows, etc. The function involving collecting statistics is related to the query optimizer and shall be discussed later in section IV-F.

Listing 2. External interface for the Table component.

```
struct TableModule {
   Module module;

   Table * (*create)(...);
   uint16_t (*get_num_cols)(...);
   DataType (*get_column_type)(...);
   int (*add_column)(...);
   void (*remove_column)(...);
   void (*insert_rows)(...);
   void (*remove_rows)(...);
   void (*update_rows)(...);
   void (*update_cells)(...);
   int (*gather)(...);
   int (*release)(...);
   int (*fetch)(...);
   TableStatistics** (*collect_stats)(Table *);
};
```

The interface definition is included in the Listing 2. We did not list all the function parameters, and instead replaced them with '...' for brevity and to avoid confusion regarding each parameter's purpose.

Both of these structures are exposed to the external implementer who can extend the Table structure with more metadata, and implement the TableModule interface by registering functions that implement the interface methods.

B. Internal Implementation The internal implementation consists of basic versions of the external interface calls, as well as some extra functions that should not be exposed to the external implementer. The internal implementation of the interface functions offloads most of the functionality decisions to the external implementation, by performing calls to the external module's functions (assuming that a module serving the role of this component has been (dynamically) loaded). For example, to insert rows into a table, the internal implementation of `insert_records` invokes the external module's matching function to perform the task.

Both the internal implementation of interface functions and the extra functions that are not exposed in the interface are responsible for performing some internal bookkeeping operations to keep track of the changes and to update other logically correlated components. For example, the Executor may call the Table's internal implementation of `insert_records` to trigger the insertion of one or more rows in a given table. The internal implementation of `insert_records` (aside from calling out to the external Table module to insert the tuples), also triggers the `index_insert` internal method for any indexes associated with this table (which in turn call out to the corresponding external Index modules, depending on index type). The Table internal implementation also has extra functions such as: `table_col_add_vls`, to trigger the tracking of a new VLS column in the VLS internals and Object Management system, `table_col_get_vls`, to retrieve such information if a column is variable-length, `table_col_create_index` and `table_col_get_index`, to allow inter-operation with the Index component framework-side internals and the core Object Management system, and other such extra internals which would otherwise compel the designer of a Table module to account for dependencies with other components. The internal functions (`table_*`) are shown in Listing 3.

Listing 3. Internal implementation of the Table component.

```
/* Internal-only functions for inter-operation with other components */
int table_column_add_vls(...);
ObjectID table_column_get_vls(...);
int table_column_create_index(...);
int table_column_get_num_indices(...);
ObjectID table_column_get_index(...);

/* Internal implementation of interface functions,
   which call out to external implementation */
ObjectID table_create(...);
uint16_t table_get_num_cols(...);
DataType table_get_column_type(...);
int table_add_column(...);
void table_remove_column(...);
void table_insert_rows(...);
void table_remove_rows(...);
void table_update_rows(...);
void table_update_cells(...);
int table_gather(...);
int table_release(...);
int table_fetch(...);
TableStatistics** table_collect_stats(Table *);
```

C. External Implementation example - ColumnStore Module We implemented a ColumnStore module that employs a column-oriented data layout, and used it in our tests for Slingshot. The external module extends the Table metadata structure, and implements (library-local) functions to serve the role of the Table interface calls. All of the external implementation functions get compiled into a shared object, which constitutes the ColumnStore module. The designer then defines a descriptor containing the name of the ColumnStore module and its location. Finally, the ColumnStore module gets loaded dynamically into Slingshot, through the Module Manager. Listing 4 briefly shows how the ColumnStore module implements the Table interface. The `cs_*` functions are the ones implementing the column store functionality.

The physical data layout for tables and columns involves a direct mapping to their OID. In our implementation of the module, a table will be stored in multiple files named $\langle OID \rangle\_\langle chunkNo \rangle$, where OID is the table's OID and $chunkNo$ is 0 or higher, depending on whether the table is large and must be split. We allow the possibility that the table be split into multiple chunks (either due to filesystem restrictions on maximum file size, or for various optimizations when faced with updates, easier maintenance of a free space map, reshuffling data for locality, etc.). Similarly, columns are stored in file(s) corresponding to the column OID. For example, let's consider a table 'Products', with 2 columns: 'productID', and 'productName', each of them fixed-length (in reality, productName may be a varchar, but we shall see later how we can handle variable-length fields). The Products table is assigned OID=1, and stored on disk as file 1_0 which contains the table metadata; productID is assigned

OID=2 and stored on disk as file 2_0 containing the productIDs; productName is assigned OID=3 and stored on disk as files 3_0 and 3_1 (let's say since product names have larger size they exceed the maximum chunkSize and spill over into a new chunk). In practice, we did not have to spill over into a new file chunk for any of our tests, but we expect with considerably larger datasets (or if a designer wants to keep the the chunksize small for various reasons) this may be common. Therefore, we allow for this possibility and adjust our implementation to account for these cases.

Listing 4. ColumnStore module implements the Table interface

```
struct ColumnStore {
  Table table;
  uint16_t num_cols;
  CSColumn columns[];
  ...
};

TableModule ColumnStoreModule =
{
  .module = { .type = MODULE_TYPE_DATA_LAYOUT,
              .supports_parallel_writes = 0,
              .name = "ColumnStoreModule",
              .package = "ColumnStore"},
  .create = (Table* (*)(...)) cs_create,
  .get_num_cols = (uint16_t (*)(Table* t, ...)) cs_get_num_cols,
  .get_column_type = (DataType (*)(Table* t, ...)) cs_get_column_type,
  .add_column = (int (*)(Table* t, ...)) cs_add_column,
  .remove_column = (void (*)(Table* t, ...)) cs_remove_column,
  .insert_rows = (void (*)(Table* t, ...)) cs_insert_rows,
  .remove_rows = (void (*)(Table* t, ...)) cs_remove_rows,
  .update_rows = (void (*)(Table* t, ...)) cs_update_rows,
  .update_cells = (void (*)(Table* t, ...)) cs_update_cells,
  .gather = (int (*)(Table* t, ...)) cs_gather,
  .release = (int (*)(Table* t, ...)) cs_release,
  .fetch = (int (*)(Table* t, ...)) cs_fetch,
  .collect_stats = (TableStatistics** (*)(Table *)) cs_collect_stats
};
```

The interface functions intuitively map to column-store semantics: create a new column store, get the number of columns or column types, add/remove a column, retrieve, insert, or update records, etc. Newly inserted records are appended at the end of the corresponding files for the respective columns. When updating a record, we overwrite the contents at the corresponding offset within the file. Removing a record marks that record in a free space map for re-use.

One interesting example of how components inter-operate is the "gather" function, invoked to retrieve a set of records specified by their rows and columns. In the ColumnStore module implementation, the Storage Manager's interface (only external Slingshot interfaces are visible from external modules) is called to serve the corresponding pages where the records reside (either on disk or in memory, if already there), or pin them in memory to avoid bufferpool eviction. The records gathered point into the memory-mapped pages at the corresponding offsets. The "release" function from the Table interface releases (unpins) the pages requested from the Storage Manager in a matching "gather" call. The "fetch" function acts similarly to "gather", except it pins the corresponding pages, copies over just the records needed and then unpins the pages immediately before returning to the caller.

Finally, note that our proof-of-concept ColumnStore module is simpler than full-fledged column stores like Vertica. Our ColumnStore module does not use compression, nor does it store columns sorted, but rather as columnar projections of a row-store. For fast search we add indexes to the appropriate columns, similar to PostgreSQL. Nevertheless, the designer is free to implement a different ColumnStore module which uses a data layout that keeps values sorted and compressed.

This concludes the description of the Table component. In summary, the Table component includes an interface, and an internal implementation that glues the Table component to other related components in Slingshot and offloads most of the functionality to the external module. The external implementation/module must extend the interface, yet is allowed flexibility in design decisions with respect to table layout on disk, column vs. row-oriented, and any various low-level programming optimizations that the designer may wish to leverage (e.g., efficient algorithms, compiler optimizations, etc.).

2) II. Variable-Length Storage (VLS): Most database engines separate the variable-length data from the main table, to avoid alignment issues and to obtain better data cache utilization. We make the same *delineation*, separating variable-length data from fixed-size data, and thus restricting the designer to abide by the same principle. A VLS column will be stored in a separate file, while the main table will just store a fixed-size header containing metadata on how to locate each individual record, and possibly some fixed prefix from each tuple for indexing purposes (for example a prefix for varchar, or the MBR for a spatial shape). As a result, each such column will use an extra OID (corresponding to the file containing the variable data), which

will be linked to the parent table. This internal decision does not assume a column-store model for the entire table, since the header and prefix for a variable-length field can be stored in either a row-store or column-store fashion.

A. Interface Similar to the Table interface, the VLS interface contains the following elements:
a) The *VLS entity* is described in a VLSManager structure (Listing 5), containing minimal information about the variable-length column (an OID, and a VLSModule containing the callbacks that implement the functionality).

Listing 5. VLS entity

```
struct VLSManager {
   ObjectID oid;
   VLSModule * vlsm;
};
```

b) *External interface*: The VLS functionality is described in the VLSModule interface (Listing 6), containing a set of function calls that handle operations on variable-length data. These two structures are exposed to the external implementer, who can extend the VLS structure and implement the VLSModule interface.

Listing 6. VLS interface

```
struct VLSModule {
   Module module;

   VLSManager* (*create)(ObjectID oid, DataType dtype);
   int (*destroy)(VLSManager *vls);
   Handle (*add_record)(VLSManager *vls, void *bytes, uint32_t size);
   void (*remove_record)(VLSManager *vls, Handle h);
   void * (*fetch_record)(VLSManager *vls, Handle h);
   Handle (*update_record)(VLSManager *vls, Handle h, void* bytes, uint32_t size);
};
```

B. Internal Implementation Similar to the Table interface, the VLS interface contains a metadata structure and a set of methods that handle operations on variable-length data. The interface calls are again intuitive: create or remove a VLS entity, add/remove a record to/from VLS storage, update a record, fetch records, etc. The internal implementation of the VLS consists of a set of functions that provide minimal functionality to glue the VLS internals to other related components (such as Table and Index), while the rest of the tasks are offloaded to the external implementer, by performing calls to the external module's functions.

Listing 7. Example of offloading functionality to an external module implementation

```
Handle vls_add_record(VLSManager *vls, void * bytes, uint32_t size) {
   return vls->vlsm->add_record(vls, bytes, size);
}
```

For example, to add a record to a variable-length column (and fill it with 'bytes' content of 'size' length), the internal implementation calls out to the respective external module to perform the task using its implementation of `add_record`, passing the 'bytes' and 'size' (see Listing 7). In keeping with the design principle of *Decouple*, the internal implementation makes no assumptions about the variable length data layout.

C. External Implementation - sample VLS Module We design an *external VLS module*, called FlexStore, that implements the VLS interface, using a fairly straightforward approach. Variable-length records are stored one after another, each preceded by their length.

Listing 8. FlexStore module - FlexStore entity

```
struct FlexStore {
   VLSManager manager; /* Contains the ObjectID, and VLSModule */
   uint64_t size; /* size of file storing varlen records */
   DataType type; /* data type for these records */
   uint64_t num_records; /* other record metadata */
   ...
};
```

As with the ColumnStore module, the FlexStore module contains a structure extending the VLSManager, and a VLSModule structure containing the pointers to functions that implement the interface calls. We include brief code snippets (Listings 8 and 9) showing how to extend the VLS structure and implement its interface.

Listing 9. FlexStore module implements the VLS interface

```
VLSModule FlexStoreModule = {
    .module = { .type = MODULE_TYPE_VLS,
                .supports_parallel_writes = 0,
                .name = "FlexStoreModule",
                .package = "FlexStore"},
    .create = (VLSManager* (*)(...)) flex_create_storage,
    .destroy = (int (*)(VLSManager*)) flex_delete_storage,
    .add_record = (Handle (*)(VLSManager *, ...)) flex_add_record,
    .remove_record = (void (*)(VLSManager *, ...)) flex_remove_record,
    .fetch_record = (void* (*)(VLSManager *, ...)) flex_fetch_record,
    .update_record = (Handle (*)(VLSManager *, ...)) flex_update_record,
};
```

B. Index Component

The Index interface is similar in design to the Table and VLS. The interface is composed of an Index structure and an IndexModule external interface, which both get exposed to the module implementer (see Listing 10).

Listing 10. Index interface

```
struct Index {
   ObjectID oid;
   IndexModule * mod;
};

struct IndexModule {
   Module module;
   Index * (*create)(...);
   Index * (*load)(...);
   void (*insert)(Index *, ...);
   void (*remove)(Index *, ...);
   void (*update)(Index *, ...);
   IndexState * (*search_start)(Index *, ...);
   uint32_t (*search_next)(Index *, IndexState * state, ...);
   void (*search_end)(Index *, IndexState * state);
   void (*display)(Index *, ...);
   IndexStatistics* (*collect_stats)(Index *);
};
```

Functions in the interface involve creating an index, loading (including bulk-load), inserting, removing, updating, etc. Additionally, for index search the interface contains methods for starting a search or continuing a search from a previous search state. The interface allows the implementer to return matches either one at a time, or in chunks in each search. The search state indicates where to resume the search (e.g., from which leaf in a B-tree was the last match retrieved and what paths have been explored so far). As with the Table component, a function exposes index statistics to the query optimizer, as discussed later in section IV-F. The internal implementation calls into the external module to handle index operations.

We implemented two sample modules that extend the Index component interface: a B-tree and an R-tree. We use the R-tree for our spatial tests, and the B-tree for our non-spatial evaluation. The index modules are responsible for the index data layout and use the Storage Manager interface to retrieve the pages that contain the desired index data.

We use similar optimizations as in our previous work [7] for the R-tree module functions. We also use our observation that the index should be restricted to only fixed-size data. Specifically, indexing variable-sized data is permitted, but should be done only using a fixed-size representation of the variable-length data (e.g., a prefix for varchars or a minimum bounding rectangle (MBR) for geometries). Since only the prefixes (rather than the full data) are indexed for a variable-length column, an additional verification may be required to ensure the records returned actually match the search criteria.

C. Storage Manager

The role of the Storage Manager is to provide a layer between the physical data stored on disk and the data processing infrastructure (the Transaction Manager and Executor). Its main attributes are to serve requests for data, to handle bookkeeping of data blocks on disk, and to provide in-memory caching of data according to a replacement policy.

Abiding by our design principles, the Storage Manager has no concept of rows, columns or tuples, and instead operates on raw data chunks called *pages*. Each page is a memory-mapped chunk from a file on disk containing data (table records, table metadata, index records, etc.). The Storage Manager keeps track of the mapping between the memory page and the disk handle (the file and offset on disk where the page data physically resides).

The Storage Manager interacts with most components, especially with the Data Layout and Index. For example, the Table module can determine (based on its internal data layout and the Storage Manager page size) in which page a given record resides and ask the Storage Manager for that page. The Storage Manager responds by returning a pointer to the requested memory-mapped page where the data resides and pins that page in memory to indicate it is in use and should not be evicted by the replacement policy. If the data is not in memory, the corresponding disk block is fetched from disk and memory-mapped into a new page. All the bookkeeping (e.g., keep track of which pages from which objects are in memory, fetch page from disk and enforce a replacement policy, make sure pinned pages do not get evicted, etc.), falls within the attributes of the Storage Manager. The Storage Manager also chooses its page size.

By providing a caching layer that takes care of retrieving data from disk, using a replacement policy, and flushing dirty pages to disk, the storage manager acts like a bufferpool in a traditional RDBMS. However, the Storage Manager also takes care of interacting with the Transaction Manager and Log Manager. Together, they ensure ACID properties and concurrency control for transactional workloads.

We make no assumptions on the types of workloads served by the data processing infrastructure as a whole. We allow for the Storage Manager to serve both analytical and transactional workloads, depending on the implementation of the interface. For example, to support read-only workloads, the interface can be used to implement the Storage Manager functionality in a read-only fashion, avoiding most of the overhead of transactional bookkeeping, version control etc. However, for OLTP-type workloads, the balance between good performance and ensuring transactional properties (especially isolation and durability level) may be left up to the implementer.

A. Interface Similar to all other previous interfaces, the Storage Manager interface defines a structure called StorageManager, and a StorageModule interface containing the set of function calls exposed to the implementer (Listing 11).

Listing 11. Storage Manager interface

```
struct StorageModule {
    Module module;
    StorageManager* (*initialize)(uint64_t pool_size);
    void (*finalize)(StorageManager*);
    TxBinding (*bind_transaction)(OidOperation* objects, uint32_t n_objects,
                                  TransactionID);
    void (*transaction_commit) (StorageManager*, TxBinding);
    void (*transaction_rollback) (StorageManager*, TxBinding);
    Buffer (*page_create)(StorageManager*, ObjectID, TxBinding);
    int (*page_remove)(StorageManager*, Handle, ObjectID, TxBinding);
    void (*forget_oid)(StorageManager*, ObjectID);
    Page (*page_markdirty)(StorageManager*, Handle, ObjectID, TxBinding);
    Page (*page_pin)(StorageManager*, Handle, ObjectID, TxBinding);
    void (*page_unpin)(StorageManager*, Handle, ObjectID, TxBinding);
    void (*flush)(StorageManager*);
    BufferpoolStatistics* (*collect_stats)(StorageManager *);
};
```

Omitting the three transactional functions (which we discuss later), the functions in the interface are as follows:

- `initialize`: initializes Storage Manager resources, locks, bufferpool, metadata on the data tables, indexes, etc.
- `finalize`: finalize Storage Manager, either on request (drop a Storage Manager and switch to another), or upon shutting down Slingshot.
- `page_create`: Creates a new page in memory for a given object. The page gets pinned in memory as well, assuming the page is created to be used right away by the caller. The caller has to unpin the page before it can be evicted by the replacement policy.
- `page_remove`: Removes a page from memory and deletes the corresponding file chunk on disk.
- `forget_oid`: Removes an entire object, by clearing all its pages from memory and deleting all the files corresponding to that object on disk.
- `page_pin`: Pins a page in memory, so that the replacement policy cannot evict it. Once a page is pinned the replacement policy has to skip it and go to the next one in line for eviction according to the replacement policy. There is a limited number of pinned pages, correlated with the maximum capacity of the bufferpool, so a `page_pin` may fail if too many pages become pinned. In practice, a pin should only be acquired for a short duration, for example while retrieving a record and a matching unpin call should follow shortly. When dealing with concurrency, multiple concurrent transactions can pin a page, incrementing a pin count atomically. For read-only transactions this is straightforward, multiple readers can simply pin a page concurrently, and the pin count will keep track of how many transactions are still using the page. We shall discuss later (Section IV-C2) how a readers/writers scenario is handled.

- `page_unpin`: Unpins a page from memory. Once a page is unpinned, it becomes fair game for eviction by the replacement policy. At a later time, when the replacement policy kicks in, it always checks if the page in line to be evicted is dirty and flushes it to disk if necessary, before fetching a new page in its place.
- `page_markdirty`: Marks a page as dirty, when the caller has modified its contents. The assumption is that the page is already pinned. A page may be pinned for reading or for writing, so the called has to specifically mark a page as dirty when it writes to the page. It is expected that a dirty page's changes are persisted to disk upon eviction.
- `flush`: Flushes all dirty pages from the bufferpool to disk, to persist changes. This provides an additional on-demand mechanism for persistence, rather than having to rely solely on the replacement policy to propagate changes to disk.

The `collect_stats` function exposes statistics related to the buffer management within the Storage Manager, as discussed later in section IV-F.

B. Internal Implementation The internal implementation of the interface methods (having the same function names as the interface methods, except prefixed with "`sm_`") enables the Storage Manager to inter-operate with other components in the Slingshot framework. For example, the internal methods `sm_bind_transaction`, `sm_transaction_commit`, and `sm_transaction_rollback` allow inter-operation with the Transaction Manager, while the `sm_page_pin/unpin`, `sm_page_create`, `sm_page_markdirty`, etc. calls allow the DataLayout and Index modules to perform operations on records by asking for specific pages, signaling that the contents of a page have been altered when records residing in it are modified, etc. As with other components, the internal implementation leaves functionality decisions to the designer and calls into the external module's counterpart methods.

C. External Implementation We implement two example Storage Manager modules, designed for different purposes. The first one, OLAPStorage, usable in data warehousing-type applications, is lightweight and avoids any transactional aspects in favor of performance. The second Storage Manager is aimed at OLTP-type workloads and provides transactional semantics.

1) C1. OLAPStorage Module: We show how we can implement a non-transactional Storage Manager (in essence, an enhanced bufferpool) on top of the Storage Manager interface, to serve for read-only analytical workloads. Our OLAPStorage module sacrifices transactional semantics in favor of performance. Since we have no restrictions on how to implement the module, we optimize our code as much as possible and make our own ad-hoc decisions on how to keep track of data in memory, what page size to use, etc. In this module implementation, we have chosen a page size of 8KB, similar to PostgreSQL.

The Storage Manager must serve pages from disk or from memory when instructed by other components, and must evict older ones when the bufferpool capacity is reached. Internally, the module keeps track of which pages are mapped in memory and their origin on disk, which pages are pinned and which pages are dirty (the 'read-only' aspect refers to the analytical queries which are mostly selects, although one-shot bulk updates at rare intervals are allowed, as with any large data warehouse; thus, pages can become dirty).

The bufferpool keeps a circular linked list of bufferpool objects that contain the necessary metadata about a page. The linked list enforces an LRU ordering. A dedicated pointer keeps track of the least recently used page in the list, in order to allow quick eviction. If the next victim happens to be pinned, the list is traversed in order to retrieve the oldest least recently used page which is not pinned. Orthogonal to this list of pages, we maintain some internal data structures for fast access to pages when requested by other components. For brevity, we omit other implementation-specific details and optimizations.

A very important aspect of a Storage Manager is the interaction with other Slingshot components. To illustrate how other modules interact with a Storage Manager, we show how the external implementation of the Table component (for example, our ColumnStore module) interacts with the `sm_page_pin` implementation from the OLAPStorage module. When the ColumnStore module needs to fetch a given record (or multiple records), it must request to pin the page(s) where the record(s) reside. The ColumnStore module is responsible for the data layout on disk, so it can calculate in which file and at which offset the record resides. It then can pass this information to the `sm_page_pin` call, in the form of a page "handle" (file id and page number within the file, where the record is located), so the Storage Manager knows which page to fetch. Records that overlap multiple pages require multiple pin calls. The call to `sm_page_pin` goes through the Storage Manager interface (the interface is the only one visible from the external OLAPStorage module), then through the internal implementation of the Storage Manager, which in turn invokes the OLAPStorage module's pin method.

2) C2. OLTPStorage Module: The transactional Storage Manager is more complex due to having to ensure transactional semantics (ACID properties). The implementation is similar in structure to the OLAPStorage module, but needs to keep track of pages on a transactional level as well. We opt for a multi-version concurrency control mechanism, similar to snapshot isolation, in which we maintain multiple versions for each object and keep track of versions at a page-level granularity. We do not claim that our implementation of the OLTPStorage module is the most innovative or the most efficient. The goal is to show how we can realistically implement this component to operate as the storage engine in the Slingshot framework. Other designers that use the Slingshot framework have the liberty to implement the Storage Manager module and tailor their implementation in any way they see fit, depending on the nature of their data processing application.

We now describe our approach and show how the OLTP implementation of the Storage Manager interface enforces transactional semantics. Transactions are classified into readers (read-only transactions) and writers (write-only/read-write transactions). The transaction type is determined by the Transaction Manager based on the operations involved in its set of queries, using information from the execution plan. The Transaction Manager also handles conflict detection and resolution and decides when to invoke the `sm_transaction_commit` or `sm_transaction_rollback` functions of the Storage Manager.

We enforce a form of snapshot isolation, in which reader and writer transactions operate on *epochs* (stable snapshots of data containing only already-committed changes, or transient (copy-on-write) snapshots for writers, which can turn stable on commit), and maintain object versions persisted through sparse files (containing page-granularity diffs). The Storage Manager keeps track of the chain of epochs, releases epochs that become obsolete (no more readers registered for that epoch), and persists data with the `sm_flush` interface method (either delayed data persistence if a write-ahead log (WAL) is present or durability guarantees are relaxed, or at commit time if we opt for a shadow-paging approach - we use the latter in our experiments).

Epochs

The reader and writer transactions operate on epochs. An epoch is essentially a snapshot of the database at a given time. An epoch can be stable (fully committed changes) or transient (uncommitted changes). Epochs keep track of which objects are being read, as well as what changes have been committed (stable epochs) or are in flight (transient epochs). The Storage Manager keeps track of the chain of epochs and their ordering.

Transactions logically operate on the latest stable epoch at the time when the transaction is scheduled for execution. We call this the transaction's ancestor epoch (please note that when a transaction commits, the latest stable epoch may not be the same as its ancestor). Reader transactions operate solely on the ancestor, and can operate in parallel. Writer transactions operate on both the ancestor and a new transient epoch, to isolate changes until they get committed. When a writer is committed, the transient epoch turns stable. Multiple writers may operate on the same transient epoch, yet in practice we have not implemented this feature, due to added complexity of tracking conflicts in the Transaction Manager.

To keep track of when a snapshot is obsolete and can be dropped, each epoch stores metadata regarding which objects are being read. Each reader adds its read set of OIDs to its ancestor epoch. A writer adds its read set to its ancestor epoch as well. When a transaction commits, it removes its read set from its ancestor. When an ancestor's read set becomes empty, that epoch can be dropped, provided it is not the latest stable epoch.

A stable epoch stores metadata on the set of written objects that incurred changes before this epoch became stable. In a transient epoch, the set of written objects contains those modified by the in-flight writer transaction associated with the transient epoch. In both types of epochs, the object changes contain the diffs (object changes, at page-level granularity, persisted on disk using sparse files) for each of the written objects, and metadata information about each object version. Object versions are orthogonal to epochs and the implementation details are discussed later.

The diffs get propagated forward into the epoch chain; the latest stable epoch always contains references to the changes that have ever been made to the object, and which have not yet been merged into the original object's file. As a result, one has to look only in the latest stable epoch to find all the changes ever made to any object, rather than traverse backwards through a possibly long chain of epochs.

Additionally, when an epoch is dropped due to having no more readers, we can simply discard the epoch since the diffs will be available in a successor epoch anyway. Please note that dropping an epoch does not free the associated object changes, which get propagated to persistent storage separately.

Object Versions

Objects such as tables, indexes or columns (in column-stores) get modified by writer transactions. The Storage Manager tracks these object changes at a page-level in a way that ensures that other transactions see correct, up-to-date and consistent information, according to the isolation level chosen by the designer. In our module implementation, we choose to take the following approach: each time an object is modified by a writer, a copy of this object is created and the writer operates on the copy until it is time to merge the copy with the base object. By base object we refer to the file-backed OID of the original object. While Epochs are logical snapshots, object versions correspond to physical copies of pages that contain changes to an object or a previous version thereof.

A *version number* is maintained for each copy. If the object has already been modified and the changes did not make it to the base object yet, then the transaction's ancestor should have the OID in its writeSet. Therefore, the previous version can be retrieved and incremented to create a new version. If however the object was never modified (not found in the latest stable epoch's write set), we simply create an object version metadata structure with a version number of 1. The copy of the object will be stored physically in a version file, named as the original OID with the upper 8 bits masked with the version number. Since OIDs are 64-bit wide, this does not impose too much of a limitation on the maximum number of objects in the system. However, this can be fine-tuned and is simply a decision in our module implementation that does not restrict others from using a different convention for dealing with object versions.

We obviously cannot create copies of entire tables for each transaction in the system, so we operate at the granularity of pages and only store the diffs in a *version file*. A version file is a sparse file - which is basically like any regular file, except

that the blocks that only contain zeros are not actually stored on disk. For example, we can create a copy of a 10GB table object that only takes up 32KB on disk, storing only the pages that are modified (the diffs) at the exact same offsets as the original pages. Consequently, merging object versions into the base object is fairly straightforward.

Object versions are chained as well, in order of version numbers. Since ObjectChanges do not get freed when dropping an Epoch, object versions are still available for reconstructing changes and merging them to the base object at a later time. We maintain a flag in the metadata structure indicating when an object version goes outside the scope of any Epoch. In our implementation, traversing the object version chains and merging them into the base object, can either be done in parallel at periodic intervals by a "compactor" thread, or on every `sm_flush` call (we pick the latter in our experiments, since it did not seem to have a significant performance impact; however, running the concurrent compactor thread may be the safest option in general, to avoid possible unnecessary overheads). Furthermore, since all changes made by writers are file-backed in versioned OID files, and the information about all OIDs (including versioned OIDs) is kept in the Storage Manager metadata, recovering committed transactional changes after a crash is just a matter of going through all objects, retrieving the diffs from the versioned files and merging them into the file corresponding to the main object.

The caller is agnostic of all the OLTPstorage internals, especially object versions and always passes the base (original) OID into functions that require an ObjectID (e.g., the `sm_page_*` functions). The methods in the OLTPstorage module can easily determine the object version based on the transaction binding number (discussed later in this section) passed by the caller and the internal metadata.

An interesting question that arises is how many object versions do we keep for each object. Technically, having sparse files mitigates storage bloating, yet we do not want a memory bloat with too much object version bookkeeping either. The number of object versions is limited to 8-bits, yet we can impose a higher or lower limit on the number of object versions, depending on how it affects performance for the target application. In practice our test workloads did not experience going over the 256 versions per object, so we did not explore this issue further and since it is not relevant for a proof-of-concept OLTPstorage module, we leave further exploration of this trade-off to module designers.

Synchronization optimizations

All shared Storage Manager resources are synchronized using either reader-writer ticket locks (similar to the ones used in most Linux kernels) for quick atomic operations (e.g., updating flags or counters), or pthread mutexes for slightly longer operations (e.g., fetching pages from disk into in-memory buffers).

Transactional interface methods

The following interface functions are left empty in the OLAPStorage module, but play an important role in the OLTPStorage module:

- `sm_bind_transaction`: Registers the current transaction in the Storage Manager. The TransactionID, objects this transaction is expected to touch, and the type of operations (read/write) for each object are passed into this function by the caller (the Transaction Manager). The Storage Manager records this metadata in its internal structures and generates a unique *transaction binding number* (TBN) that the caller can use as a unique identifier for this transaction in future Storage Manager calls. Storage Manager bookkeeping pertaining to epochs that this transaction operates on is also mapped to the TBN. In the framework, the choice of TBN is left to the module implementation. In our OLTPstorage module, the TBN is the address of the struct containing the Storage Manager's metadata on that transaction.

Epoch management operations necessary before transaction execution are also handled here: readers and writers obtain the latest stable epoch and set it as their ancestor, as well as add their read sets to the ancestor, writers get assigned a new transient epoch and add their write set to it, create all other appropriate structures, etc.

- `sm_transaction_commit`: Based on the TBN, we can determine the transaction type (R/W). The committing transaction's read set (objects) is removed from the epoch that the transaction operates on, and an epoch retirement decision is made. For writers, the page-level diffs for objects in the write set are merged with the ones from the latest stable epoch, sometimes creating a new stable epoch (if the current stable one still has readers registered). Conflict detection is handled orthogonally at a higher level by the Transaction Manager.

The durability property can be ensured by flushing all the writer's diffs to disk into the corresponding versioned file(s). This ensures that before returning to the caller, all changes have been propagated to persistent storage. This property can be relaxed by deferring flushing to a later time. However, if a WAL is present in the Transaction Manager module implementation, a data flush can be safely deferred regardless.

- `sm_transaction_rollback`: A rollback only happens in case a conflict is detected by the Transaction Manager. The Transaction Manager's `transaction_rollback` method takes care of its internals and then calls the Storage Manager's `sm_transaction_rollback` method so the Storage Manager can do its own cleanup as well. A transaction rollback for readers acts essentially the same way as a commit, since no changes are made to the data. Writers simply drop their transient epoch and remove their read set from the ancestor epoch.

The interaction between the transactional functions and the Transaction Manager are limited to TIDs/TBNs and are described in Figure 3. All other Storage Manager functions (e.g., `sm_page_pin`, `sm_page_markdirty`, etc.) take into account (using the

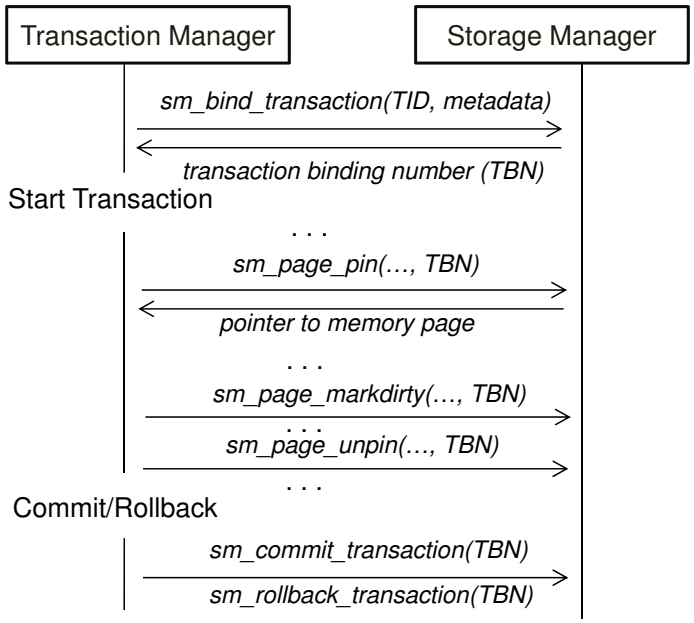

Fig. 3. Transaction Manager calls into transactional Storage Manager.

TBN generated by bind) that requests for pages can originate from within different transactions, operating in different epochs, on different object versions.

D. Executor and Transaction Manager

The Executor handles incoming queries that have been parsed and optimized into an execution plan (a set of operations to be executed through the framework). Since Slingshot does not have a query parser and optimizer yet, each query is manually implemented as a sequence of operators (using interface calls to the right components) that a query planner would normally generate as its execution plan. The Executor receives each transaction as a sequence of such query plans.

The Executor operates at the level of transactions and calls into the Transaction Manager to enqueue new transaction jobs. A transaction job contains the execution plans of its queries, various query-related metadata, and also specifies its read and write set (the tables and columns read or written), so the Transaction Manager can decide if it is runnable. For non-transactional workloads, each query is enclosed in a transaction, the transaction metadata skips anything related to read and write sets, and the Transaction Manager is flagged to treat transactional operations (such as bind, commit, rollback) as no-ops.

The Transaction Manager component of the Executor schedules incoming transactions for execution, and takes care of conflict detection and resolution. The interface stores only minimal information about enqueued transactions, necessary to achieve these functions, while the rest is being handled by the Executor. The interface exposed to the module implementer contains a set of functions, which:

a) Create a new Executor and initialize it;
b) Enqueue a transaction for execution;
c) Commit a transaction;
d) Rollback a transaction;
e) Request status information on a given transaction.

Module Implementation Our Executor implementation maintains a pool of threads ready to execute incoming transactions. Arriving transactions are enqueued in a global queue. Transactions that do not conflict with any in-flight transactions (no expected conflicts, based on the set of objects they touch), are deemed *runnable*. The global queue is an ordered *nested queue* that enables efficient storage and fast retrieval of runnable transactions (similar to a skiplist). After a commit or rollback, the Transaction Manager must refresh the global queue by determining if other transactions become runnable. As transactions become runnable, they get promoted to the runnable queue while still respecting the arrival order.

The Transaction Manager module considers transactions runnable only if a) it is a reader, or b) if it is a writer whose write set (columns) does not conflict with any other writers in flight. This policy is fairly simple and avoids having to track conflicts between transactions during their execution. However, it is not the most efficient since it causes some unnecessary serialization for writers. For example, if two writer transactions have column A from table X in their write set, only one transaction is runnable, even though the writers may be touching completely independent rows. Row-level locks and an enhanced transaction

module (which takes care of conflict detection and resolution) may improve throughput by allowing more transactions to run in parallel. For our immediate goal of prototyping Slingshot and showing how a database can be implemented on top of the framework, we keep the module implementation simple, at the cost of some throughput (evaluated in Section VII-C).

The functions dealing with transaction commit/rollback also invoke the Storage Manager to take care of its own internal bookkeeping. We discuss the interaction between the Transaction Manager and the Storage Manager in Section V.

E. Log Manager

The log manager is responsible for keeping a history to track operations performed on the database, allowing for recovery in case of a crash by replaying the log. Logging makes sense in a transactional context, but technically can be dropped in case of OLAP-type applications or read-only workloads.

Given the strong interconnection of logging, transactional operations, and various forms of isolation, decoupling the logger from the Storage Manager and Transaction Manager is a daunting task. As a result, in the current version of Slingshot, we opted to place the physical logging operations in the Log Manager, while leaving the semantics of log records inside the Transaction Manager and to some extent, the Storage Manager. Essentially, the log operations simply involve writing a sequence of bytes, and the Log Manager is oblivious to the meaning of log records. The Transaction Manager and Storage Manager components are responsible for different durability and recoverability aspects. The Transaction Manager is tasked with persisting transactional operations (the write-ahead log), since it is also the one responsible for conflict detection and resolution. The Storage Manager is tasked with ensuring data durability by flushing pages of committed transactions to disk (or versions thereof, as per the isolation policy).

In future versions of Slingshot, we plan to explore the possibility of decoupling the log semantics into a separate component as well, and the implications of such design decisions in terms of modularity and performance.

F. Query Optimizer

The Query Optimizer (or Query Planner) is responsible for generating an optimal (or near-optimal) query plan, using a cost model and complex heuristics. In general, query optimizers employ a cost model based on a collection of statistics. The optimal execution plan is correlated with the following parameters: a) the selectivity of each predicate in the query; b) the combined selectivity when chaining multiple predicates; c) the order of relational operators (scans, joins, etc); and d) the CPU and I/O costs of processing each operator. As such, statistics involve both underlying system information (such as disk speed, CPU utilization, available memory etc.) as well as detailed statistics on the tables and indexes in a query. The former can be easily determined externally without interacting with other engine components (except for the available memory which requires input from the bufferpool), while the latter involves maintaining histograms for keeping track of data distribution in order to estimate selectivity more accurately. As data changes, statistics must be updated periodically (either on user request or as a background process). This implies that the Query Optimizer must be tightly integrated with the rest of the engine, to access the necessary statistics.

Our goal is to keep components decoupled, while also providing the Query Optimizer with all the relevant metadata. As a side-effect of the *Delineate* and *Decouple* principles, the statistics needed by Slingshot's Query Optimizer must be collected from several components, each responsible for a specific piece of metadata. These components are Table, Index, VLS, and Storage Manager.

a) The Table component is responsible for gathering only the metadata pertaining to its design boundaries. The Table component has a dedicated interface method (`collect_stats`), which collects for each column the following information: data type, column type (fixed or VLS), average records per page, whether records are kept in sorted order, a histogram containing the distribution of its values, the number of indexes on the column and their ObjectIDs (used to then retrieve Index metadata). The external Table module implements this method to collect only partial statistics (those pertaining to data distribution), while the index information is collected in the interface method's internal implementation (framework-side) which has knowledge of the presence of indices on the Table columns. The filled metadata structure is passed to the Query Optimizer when it invokes the `collect_stats` method.

b) The Index component contains a similar `collect_stats` interface method for gathering metadata about a specific index, such as tree depth, node count, number of memory pages used, etc.

c) The Storage Manager component has knowledge of the bufferpool utilization, therefore its interface's `collect_stats` method retrieves the amount of bufferpool pages in-use, the number of pinned pages and the total bufferpool memory.

While initially the VLS component also had a similar `collect_stats` method as the Table component, this interface method was dropped from the original design. Since we do not allow indexing variable-length data (to avoid "detoasting"-type overheads), indexes on VLS columns are generated using solely a fixed-size "prefix", which is stored along with the main data in the Table layout anyway.

These interface methods for collecting database-specific statistics, along with external statistics on system resource availability provides the Query Optimizer with the necessary information to generate an adequate execution plan. Consequently, Slingshot's

design principles and modular approach do not affect the Query Optimizer's ability to retrieve the relevant statistics and take informed query optimization decisions.

The types of statistics collected from each component are comparable to most query planners in relational engines. Nevertheless, as the field of query optimization evolves, better planners may require additional information from one or more components. In Slingshot, this is easily achievable by extending one or more Statistics data structures (for the appropriate components), in order to relay the extra information to the query optimizer. While we acknowledge that this is semantically a cross-cutting dependency, in that the query optimizer module and, for example, the Table module, need to both be aware of the purpose of the statistics data structure extension, this is a minimal dependency and does not significantly affect the Decoupling principle.

V. Cross-cutting concerns

Slingshot is designed to minimize cross-cutting concerns and encapsulate them in the interfaces so that they can be handled by the framework, not by the implementer-defined external modules. For example, the Table module is only responsible for fetching and storing records (according to its implementer's choice of data layout), while the Index module is solely responsible for index-related operations. This design contrasts sharply with MySQL's pluggable storage engines, where the index code is intertwined with the storage manager. By decoupling components cleanly, Slingshot makes it much easier to implement each module without knowing anything more than the interfaces exposed by Slingshot.

We identify and address some semantic cross-cutting concerns between the Storage Manager and the Transaction Manager. Again, our Storage Manager differs from what MySQL or other DBMSs mean by this term. The goal of the Storage Manager is firstly to cache data so that expensive disk accesses can be avoided, so it must support caching and writeback. However, it also supports versioning control at the page level.

In Slingshot, a transactional Storage Manager must know which transactions are accessing a given page, so that it can enforce isolation. The type of isolation should not be handled by the core framework, because we want to avoid imposing a strict isolation type, or even any isolation at all (in case no transactional semantics are desired). The Storage Manager is solely responsible for serving pages and creating copies where necessary (depending on isolation policy), but not dealing with transaction conflicts. The Transaction Manager component is responsible for the conflict detection policy (optimistic vs. pessimistic, early vs. commit-time detection, etc.) and deciding when and which transactions to rollback. The Transaction Manager also offloads these decisions to the module implementation.

Consequently, the isolation policy in the Storage Manager logically meshes with the conflict detection and resolution in the Transaction Manager component. This is an important dependency which, if designed improperly, could make component decoupling a nightmare (a Storage Manager module designer would have to know what the Transaction Manager designer is doing). In most databases, this dependency causes cross-cutting concerns that result in tangled code and no easy way to change some of the transactional semantics.

In Slingshot, we limit this dependency to simply passing globally unique IDs - the initial TID (in the bind call), then the TBN generated by the bind call. All other Storage Manager interface functions have as a parameter the TBN of the transaction asking for a page, so the Storage Manager knows exactly which page version to fetch (or pin, mark_dirty, etc.), by looking up its internal metadata. The Transaction Manager is also aware of these TBNs (mapped to TIDs), it tracks finer-grained row-level data accesses, and launches the `transaction_rollback` function, according to its conflict resolution policy. Its framework-side `transaction_rollback` function stub calls into the Storage Manager's framework-side `sm_transaction_rollback` function (using the right TBN), which in turn calls the Storage Manager module's `sm_transaction_rollback` method, to drop all internal page version metadata associated with the aborting transaction.

This strategy of passing global TIDs/TBNs between components and each component keeping only the necessary internal bookkeeping that pertains to the component's functionality means we have *no true dependencies* between the two components (one component's implementation having to access internals from the another component's module). However, there are *semantic cross-cutting concerns* between the two, in that the designers can only pick a pair of pre-existing Transaction Manager and Storage Manager modules that synergize well, or design new modules keeping this semantic constraint in mind. Nevertheless, this is still much easier to deal with than in standard DBMSs, where transactional semantics and concurrency decisions permeate multiple source files associated with different logical components.

VI. Limitations

Slingshot was designed to be flexible, in the sense that it can be tailored to a specific application type, by plugging into the framework the right modules for the job. The distinction that must be made is whether Slingshot can be morphed to serve a completely different type of workload in a static or dynamic way, and to what extent. Although dynamic adaptability is not a design goal for Slingshot, it is important to clarify what the inherent limitations are. In other words, it is important to clarify whether Slingshot is a framework that, once the designer plugs in the right modules, will act as a database engine that can only be used for that specific type of application (e.g., OLAP, OLTP, etc.), whether it can serve multiple purposes simultaneously,

by simply adding alternative modules in the same Slingshot instance which operate completely independently, or whether the modules serving different purposes for the same component can also interact.

Slingshot is designed to operate in a fully dynamic fashion, by allowing loading and dropping modules at runtime, and does not prohibit the simultaneous presence of multiple modules implementing the same interface (for example, multiple indexes, multiple tables with various data layouts (row-store, column-store, etc.), and even multiple storage managers). For the purpose of the experiments we conduct, we simply configure the modules we need from the start and no adaptation is necessary for the duration of the experiment.

Nevertheless, in practice the co-existence of multiple modules for the same component, although supported, must be carefully considered. For example, having multiple types of indexes, even on the same columns, is fairly straightforward. However, the presence of multiple data layouts in the same Slingshot instance has more subtle implications: if the user wishes to store the same table in two different data layouts simultaneously, this is possible by storing two different tables, one for each data layout module. However, the Slingshot framework does not support automatic propagation of updates into both data layouts, nor can the modules inter-operate (in keeping with the Decouple principle). So instead, two queries must be executed to perform updates for each table version. Similarly, two distinct storage managers can co-exist and serve pages for distinct tables independently from each other, but inter-operation between storage managers (like migrating tables between storage managers, or serving multiple pages for a specific table from multiple storage managers simultaneously) is not something Slingshot was designed to do. Finally, for some components (like the Executor), having multiple modules implementing the same interface does not have foreseeable applications, so we did not consider this possibility in our design. In such cases, having completely separate instances of Slingshot would make more sense.

VII. Experimental Evaluation

We demonstrate experimentally that Slingshot is capable of high performance (whether latency or throughput) for various types of workloads. We use a regular off-the-shelf desktop machine with an 8-core Intel Core i7-2600 @3.4GHz, 8GB of RAM and a 7200rpm HDD, with a sequential disk read speed of ˜110MB/sec. The machine runs a 64-bit Linux OS, 3.2.0-60-generic kernel.

We evaluate 3 scenarios: a spatial analytical workload, a set of non-spatial OLAP queries, and a non-spatial OLTP workload. In all cases, we plot the average of 5 runs (we omit error bars since the standard deviation was negligible). PostgreSQL is configured with a large bufferpool (4GB), and other tuning parameters for performance. For Vertica, we used the out-of-the-box configuration and applied the disk scheduler and readahead optimizations from the installation recommended settings.

For Slingshot, testing a client-side workload is simply a matter of instantiating the Slingshot core framework, loading the right modules, and creating a unit test that sends one or more transactions (their plans) to the Executor.

A. Spatial workload

The spatial workload uses a complex analytical query from the Jackpine spatial benchmark [12] and the TIGER spatial dataset [13] for the state of Texas. The query involves an all-pair spatial join for the edges dataset (˜6 million Polyline shapes) containing the network of roads and rivers in the state, and determines the placement of bridges by evaluating an ST_Intersects spatial operation. Spatial queries use a spatial index (generally an R-tree variant) for the initial filtering step, which uses the minimum bounding rectangles (MBR) of shapes to reduce the search space to a smaller candidate set. The second (refinement) step performs complex computational geometry algorithms to determine the exact matches.

Slingshot loads our OLAPstorage module and the VLS module to store the variable-length spatial shapes. We also implemented an R-tree index module, to handle the filtering step efficiently. We compare Slingshot with PostgreSQL/PostGIS. PostgreSQL makes use of the GiST framework to model an R-tree variant in PostGIS, and its query planner picks an indexed nested loop join for the spatial join. Slingshot uses the same type of join.

As noted in Section II the generality of the GiST framework comes with a high performance cost over a custom R-tree solution. Slingshot's Index component draws the abstraction line differently, capturing what operations an index should be doing rather than how operations should be performed. As a result, Slingshot's R-tree module implementation does not incur the same overhead as the GiST. Figure 4 b) and c), shows that Slingshot outperforms PostgreSQL by a wide margin, running the spatial filtering step in roughly 16 seconds, compared to ˜20 minutes for PostgreSQL. Additionally, the full spatial join (filtering + refinement) takes just under 170 seconds in Slingshot, compared to ˜37 minutes in PostgreSQL.

We also evaluate HadoopGIS [14], a custom spatial data processing system implemented using MapReduce. HadoopGIS allows intra-query parallelism by partitioning the data efficiently and using several mappers and reducers to process the records in parallel. We configure HadoopGIS to use 8 workers (for mappers and reducers) on our 8-core system. Although this is not an apples-to-apples comparison, since HadoopGIS benefits from parallel processing, it helps to show how the Slingshot prototype performs against a custom spatial processing engine. HadoopGIS takes over 4 times longer than Slingshot to complete the full spatial join.

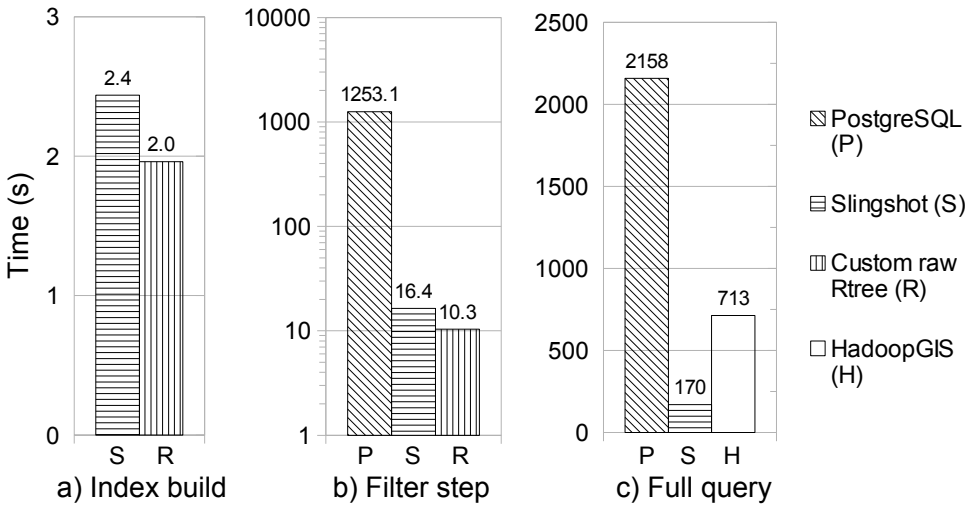

Fig. 4. Performance for spatial queries. a) R-tree build time. We leave out PostgreSQL which takes considerably longer than others. For HadoopGIS indexes are not applicable. b) Spatial Filtering. Note the logarithmic scale on the Y axis, due to the large discrepancy in values. c) Query execution time for the large spatial join (including refinement step). Custom R-trees are not included in this evaluation, since they apply only to the filtering step.

Finally, we compare Slingshot's index build time and spatial filtering performance against a raw R-tree implementation [7]. We implemented the Hilbert-based tree build strategy from our previous study [7] for both the custom R-tree and Slingshot's R-tree Index module, due to its superior overall performance. Both implementations benefit from the same code-specific optimizations, however, queries using the custom R-tree are manually implemented and will not incur the overheads of going through an Executor, requesting data from a Storage Manager, etc. Our results (Figure 4a) show that Slingshot adds very little overhead to the spatial index build time (2.4 seconds in Slingshot vs. 2 seconds for the custom R-tree). The filtering step incurs a 60% overhead with Slingshot (Figure 4b) compared to the raw Hilbert R-tree, but this is a minor cost compared to the overhead of the GiST in PostgreSQL. We did not include the custom R-tree in the full query results since it does not play a role in the refinement.

Overall, Slingshot outperforms the standard spatial DBMS, with index performance very close to a custom-designed R-tree. Slingshot exploits the freedom to optimize the R-tree module as much as a custom stand-alone implementation, while incurring minimal overhead due to calls to other components (such as the Storage Manager and VLS), which allows Slingshot to behave like a full-fledged data processing engine.

1) Breakdown by logical components - side-by-side comparison: To gain a deeper understanding of where the performance discrepancies are between PostgreSQL and the spatial database engine built using the general Slingshot framework and the dedicated external modules, we show a breakdown of where time is spent, by component. Although PostgreSQL is not modular and it is hard to clearly delimit components, we profile the function callgraph for the join query execution, and categorize the calls based on their functionality, into several logical categories that roughly resemble in scope Slingshot's components. To avoid confusion, we refer to these as *categories* or *aspects*.

We show side-by-side the breakdown of execution time spent in each of the PostgreSQL categories as well as the Slingshot components, with a few caveats. First, the Storage Manager in PostgreSQL includes everything related to data layout, fixed or variable-length storage, and bufferpool. In Slingshot, the OLAP Storage Manager is only responsible for buffering data and serving pages, whereas other aspects belong in the Table and VLS components. Additionally, one must keep in mind that these breakdowns include percentages that are relative to different baseline total execution times. The side-by-side comparison is not meant to detail all the intricate code-level optimizations that make Slingshot faster, but rather identify in which logical categories Slingshot is able to tailor a module to a specific workload, and what generality constraints (that affect performance in PostgreSQL) are dropped.

As can be seen in Figure 5, Postgres spends an almost equal amount of time in filtering and refinement, while Slingshot's filtering is much faster than the refinement stage. PostgreSQL spends overall a lesser percentage of time in actual computations (GEOS and PostGIS spatial function evaluations and geometry computations), and relatively more time in memory context management, de-TOAST-ing data and GiST support overhead. These discrepancies are caused by some design decisions for generality (e.g., variable-size index entries, GiST restrictive abstractions for index operations, etc.) which end up hurting performance, as shown in [15]. The key idea is that although we design the Slingshot framework to be general, the modules are specifically implemented for optimized spatial workload support and eliminate some unnecessary generality constraints that degrade performance, in order to behave like a custom-tailored solution.

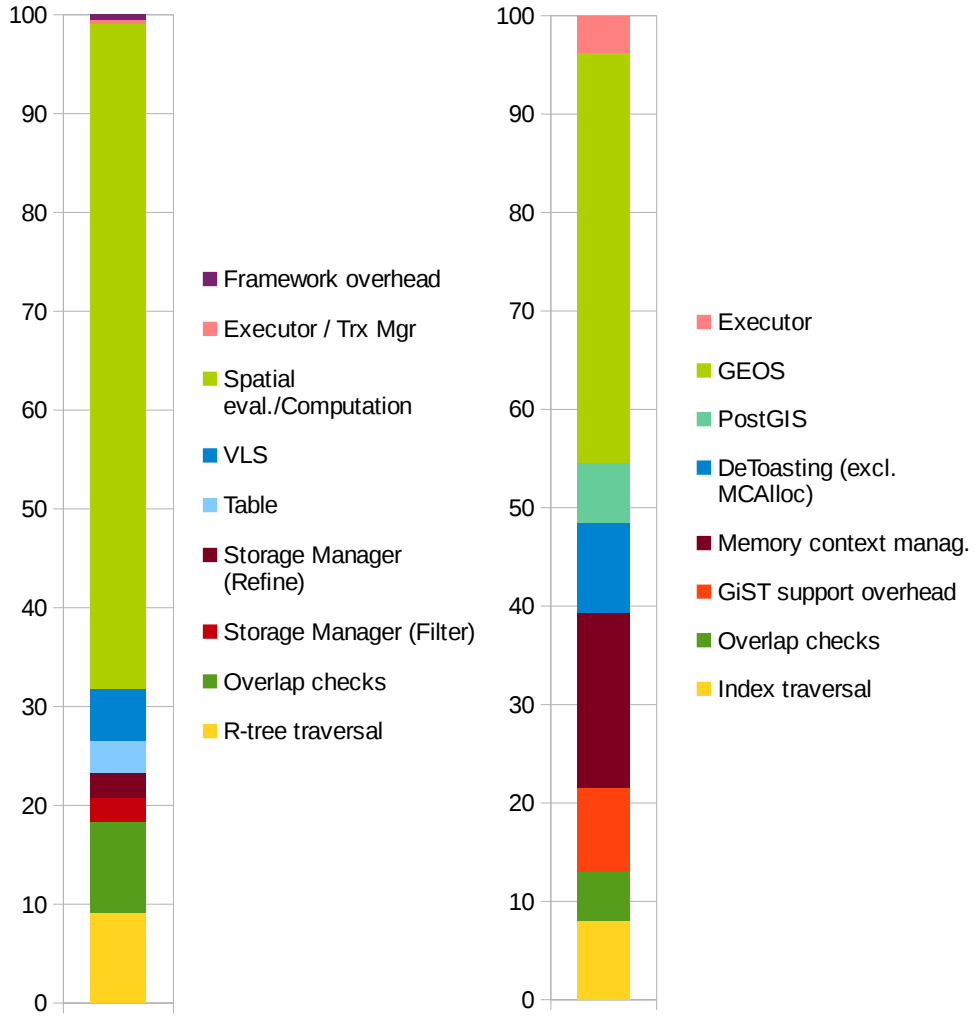

Fig. 5. Comparison of relative time spent in each logical category, for Slingshot, and PostgreSQL respectively.

B. Non-spatial OLAP workload

For the non-spatial OLAP workload, we use datasets from the TPC-H benchmark at two different scale factors: - SF1 (1.5 million records in the Orders table) and SF10 (15 million Orders).

We evaluate (a) time to create a B-tree index on order_date, (b) a query that uses index scans, to test the efficiency of our B-tree module implementation ("Select count(*) from orders where o_orderdate = '1996-01-02'::date;"), and (c) a more complex join query. Slingshot does not yet support SortBy or GroupBy operators, so we could not test with the full TPC-H queries. The Slingshot design does not limit our ability to add these operators, it simply requires additional implementation effort.

We evaluate Slingshot against a traditional RDBMS (PostgreSQL) and a custom solution dedicated for analytical workloads (the Vertica column-store). Figure 6 shows the results on the smaller TPC-H dataset (1.5 million records in the Orders table). Slingshot is 7 times faster than PostgreSQL for building a B-tree index (Fig. 6a). Vertica does not use indexes; instead, it stores each column sorted on the values for fast access, and uses compression to reduce I/O. In contrast, the ColumnStore module for Slingshot neither sorts the columns nor uses compression; instead Slingshot uses the B-tree module to index the key columns, similar to PostgreSQL.

The index scan, or column scan for Vertica, also favors Slingshot (Fig. 6b), which performs the query in roughly 0.3ms, compared to 4ms for PostgreSQL and over 10ms for Vertica. Slingshot is also roughly 4x faster than PostgreSQL and over 2x faster than Vertica for the join query (Fig. 6c). For fairness, we "persuaded" the PostgreSQL query planner to use an indexed nested loop join rather than the sort-merge or hash-join plans, which were slower for this test.

Figure 7 shows the same queries on a 10x larger dataset (15 million Orders). As before, Slingshot is significantly faster (8.25x) than PostgreSQL when building the B-tree index. Slingshot is over 350x faster than PostgreSQL when scanning the

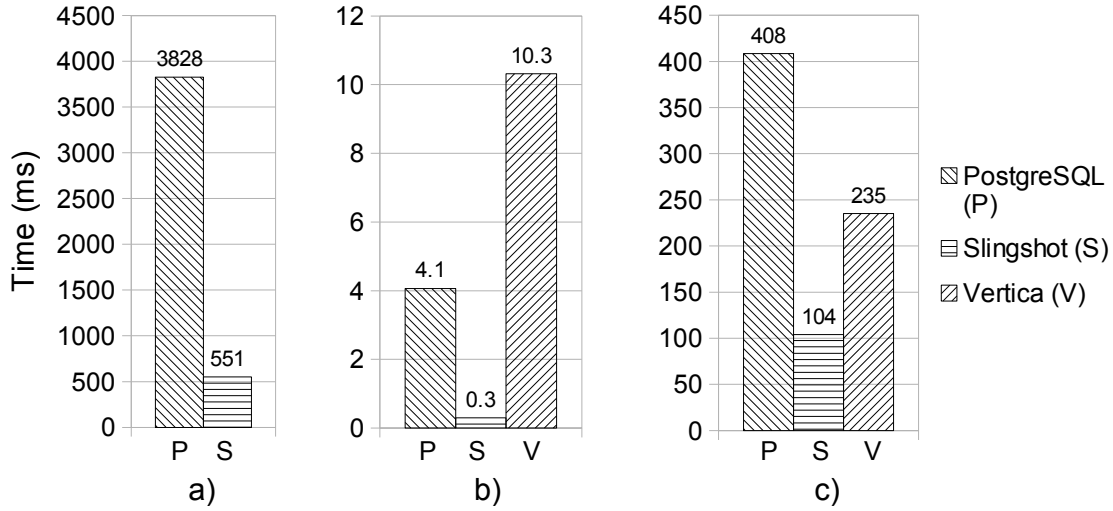

Fig. 6. Performance on OLAP workload, TPC-H dataset with scale factor 1. a) Query that creates a B-tree index on order_date; b) Query that determines the number of orders on a given date. PostgreSQL and Slingshot use their B-tree index, while Vertica takes advantage of its column-store optimizations. c) Join query. Both PostgreSQL and Slingshot use an index nested loop join, while Vertica takes advantage of its sorted columns and uses a merge-join.

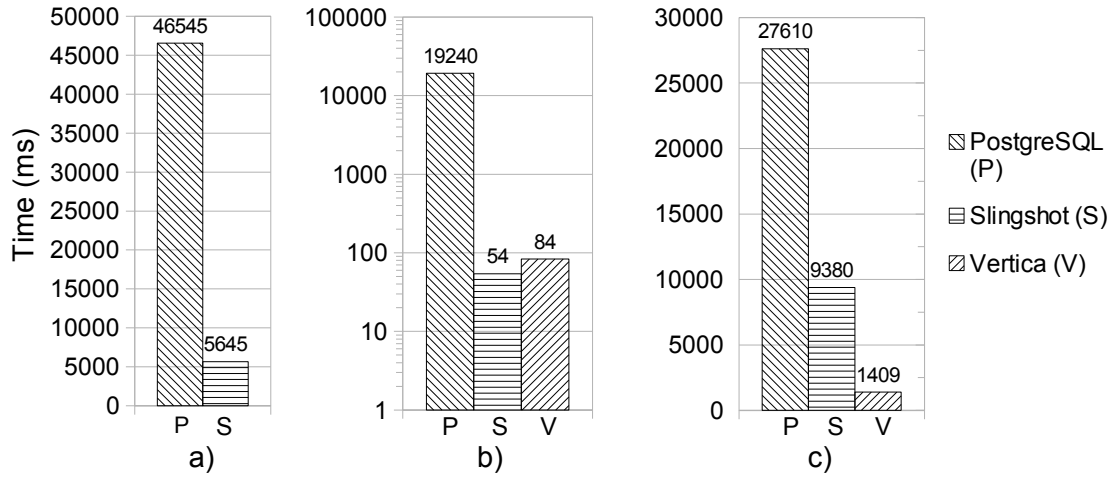

Fig. 7. Performance on OLAP workload, TPC-H dataset, scale factor 10. a) B-tree index creation; b) Query that determines the number of orders on a given date. Note the logarithmic scale on the Y-axis. c) Join query, same as with the small dataset.

large dataset for orders placed on a given date, but Vertica is quite close (only 50% slower than Slingshot). In figure 7), we use a log-scale, due to the large gap for the values between PostgreSQL and the other two engines. For the join query, Slingshot is 3x faster than PostgreSQL, but much slower than Vertica (˜9.3 seconds, compared to 1.4 seconds).

Vertica outperforms Slingshot on the large dataset because our proof-of-concept ColumnStore module is less complex. First, Vertica stores values in each column sorted, affecting searching and disk I/O, compared to Slingshot which stores the records in insertion order and uses a B-tree for fast access. Second, Vertica stores columns compressed, for reduced disk I/O. Finally, Vertica uses a mergejoin, which is very efficient due to the inputs being pre-sorted on the join column. We expect that if designers implement a more advanced ColumnStore module which supports compression and sorted columns, Slingshot would perform closer to Vertica on this case as well.

Although our Slingshot framework and sample modules are not as mature as PostgreSQL, or as specialized for OLAP workloads as Vertica, Slingshot performs better in most cases due to allowing the freedom to optimize custom code at the module level, backed by a lightweight core processing engine. With additional implementation effort to add more features in the external modules, Slingshot could match the performance of custom engines like Vertica in the remaining cases.

1) Breakdown by logical components - side-by-side comparison: As for the spatial workload breakdown, we conduct a similar analysis of time spent in various Slingshot components and their roughly-equivalent PostgreSQL logical categories for the non-spatial queries. Figure 8 shows the breakdown for the index search query on both Slingshot and PostgreSQL. Most of the execution time for both is spent in the Storage Manager (and Table, which is equivalent in functionality with some parts

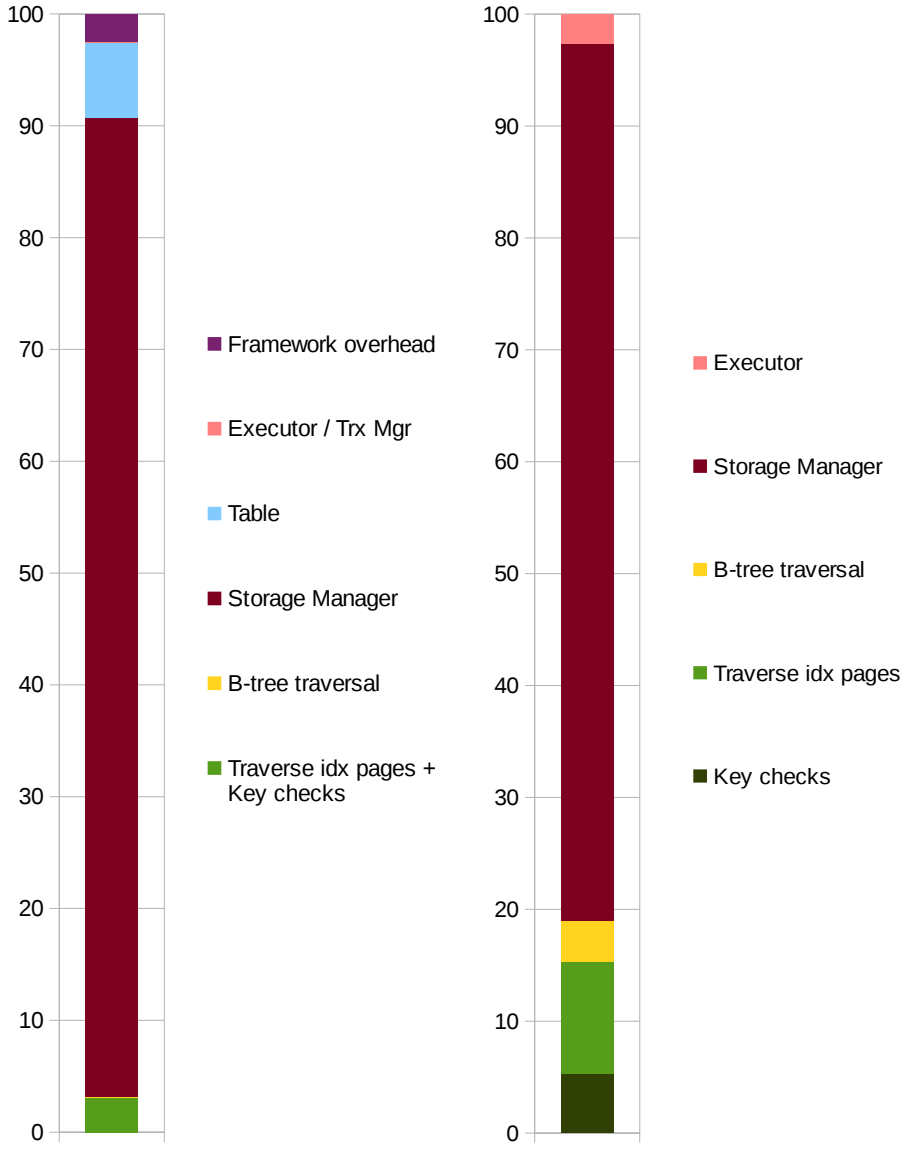

Fig. 8. Comparison of relative time breakdown for Slingshot, and PostgreSQL respectively, for the index search query.

of the Storage Manager category in PostgreSQL). The interesting part is that there is relatively more time spent in the B-tree index in PostgreSQL than in Slingshot. This is even more evident for the join query shown in Figure 9.

Based on deep inspection of PostgreSQL code, the main reason we identified is that PostgreSQL provides a generic B-tree index, able to handle any type of key and any type of indexable data, while Slingshot's B-tree module makes simplifying assumptions based on the target OLAP data and workload. First, aside from keys, pointers and some metadata, PostgreSQL adds extra metadata in each index node entry (e.g., bitmaps for optimizing multi-column indexes). In contrast, the Slingshot B-tree module assumes no NULLs are present and that multi-column indexes are limited in the number of columns. As a result, due to less metadata in each entry, we can fit more entries per internal tree node (which is a fixed page size of 8K, just like in PostgreSQL), which results in a larger tree fanout than PostgreSQL, and reaching a leaf node faster. Although it has more entries per node, internal node searches do not significantly increase due to the logarithmic complexity and the fact that the index is in memory anyway. Secondly, key checks are more complicated in PostgreSQL, which supports variable-length index entries. We did not find variable-length index entries to have much (if any) applicability, so we restricted the B-tree module to fixed-size index entries, to optimize for search speed. Finally, PostgreSQL uses reader/writer locks to ensure high concurrency for its B-trees. For an OLAP workload, incurring at least the penalty of read locks during tree traversal is entirely unnecessary, so we strip this out of Slingshot's OLAP-targeted B-tree module.

Overall, the key point is that in Slingshot the designer can afford to take liberties with specific module implementations and

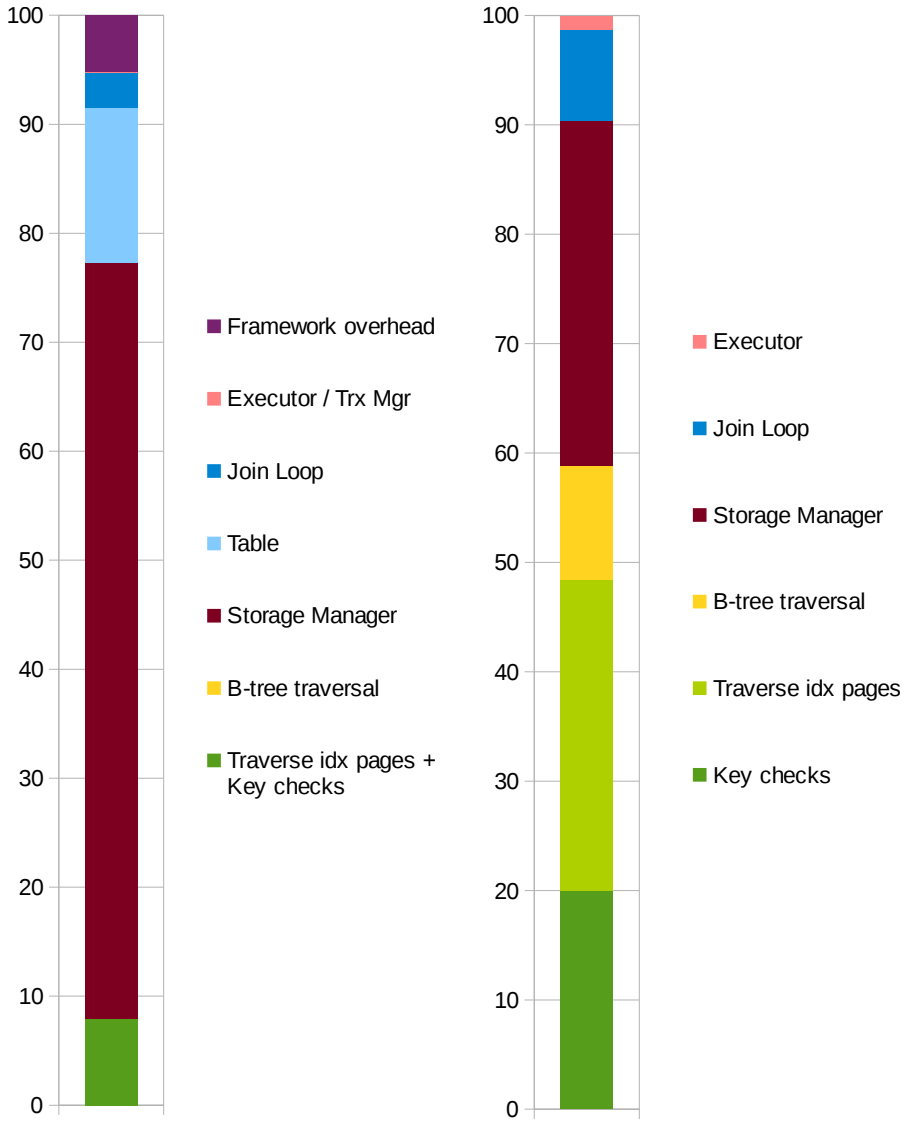

Fig. 9. Comparison of relative time breakdown for Slingshot, and PostgreSQL respectively, for the join query.

to optimize for a target workload, whereas for PostgreSQL such changes have deeper dependencies in the codebase, and are not possible without large structural changes.

C. OLTP workloads

We analyze two transactional workloads: a micro-benchmark that tests concurrent reader and read-write transactions, and a payment query from the TPC-C benchmark. For PostgreSQL, we use the jTPCC [16] implementation of TPC-C, to generate the transactions, launch the JDBC connections, and measure throughput. For the microbenchmark, we extend jTPCC to allow custom query execution and create additional tables. Since Slingshot does not have a JDBC driver, we run both the microbenchmark and the TPC-C benchmark locally, by implementing unit tests to create the workloads.

1) Microbenchmark: The purpose of the microbenchmark is to analyze the effect of serialization in Slingshot when launching concurrent reader (RD) and reader-writer (WR) transactions that execute fairly simple queries. Slingshot serializes writers that have the potential of conflicting at a column-level (as discussed in Section IV-D), while PostgreSQL allows parallel writers, detects conflicts, and performs rollbacks if necessary.

The microbenchmark simulates a simplistic online store scenario. It consists of only one table, “Products”, containing the fields ProductID, Price, QuantityInStock, CategoryID (books, electronics, etc..), and ProductName. We populate the table with 1 million records and load the data in Slingshot and PostgreSQL. The ProductID serves as a primary key and a B-tree index is created on the CategoryID. There are two types of transactions: RD (reader transaction) and WR (writer, or read-write transaction). The RD transactions simulate customers browsing the online store by category (a select operation that uses the

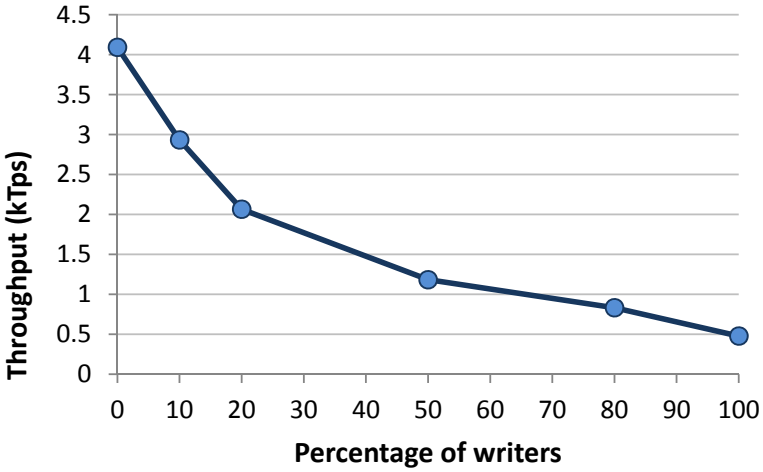

Fig. 10. Throughput variation as writer ratio (serialization) increases.

index on the categoryID to retrieve items from a given category). The WR transactions simulate customers browsing the store (select operation) and purchasing an item (update operation on QuantityInStock).

Figure 11a) shows the throughput (transactions per minute, Tpm) for both PostgreSQL and Slingshot, with a 10:1 RD to WR ratio. Slingshot achieves ˜3000 Tpm compared to only ˜2400 Tpm for PostgreSQL. In a low-contention scenario, the serialization of WR transactions does not affect throughput significantly and Slingshot's ability to optimize modules independently leads to better overall performance.

Since the microbenchmark gives us much easier control over the ratio of writers to readers, for completeness we checked how throughput is affected when the percentage of writers increases. As can be seen in Figure 10, throughput decreases with an increasing proportion of writers, due to transaction serialization.

Next, we test Slingshot on a query from the TPC-C benchmark with much higher contention, analyze how Slingshot compares against PostgreSQL, and discuss the serialization problem further.

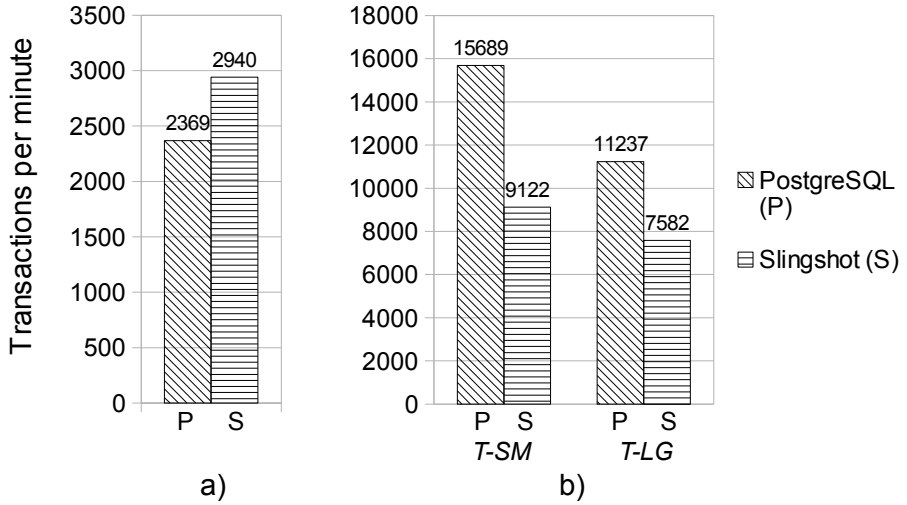

Fig. 11. Throughput comparison for OLTP workloads for Slingshot and PostgreSQL. a) Transactional microbenchmark; b) TPC-C benchmark, Payment transaction, on a small (T-SM) dataset and a large (T-LG) dataset.

2) TPC-C: We compare PostgreSQL and Slingshot on the industry-standard TPC-C benchmark. Implementing all of the TPC-C transactions manually in Slingshot is an onerous task in the absence of a query planner module, so we only implement the Payment transaction from TPC-C, which involves both Selects and Updates to each of the tables (Warehouses, Districts, and Customers).

To achieve a fair comparison (in terms of selectivity), we use the same query parameters for both PostgreSQL and Slingshot. We first run PostgreSQL using the jTPCC benchmark driver and record the trace (the values used as parameters for the transactions). We use this trace to parameterize the transactions in Slingshot with the same values. We use two TPC-C datasets

to show how scale affects performance: i) T-SM (1 warehouse, 10 districts, 30K customers), and ii) T-LG (100 warehouses, 1K districts, 3M customers).

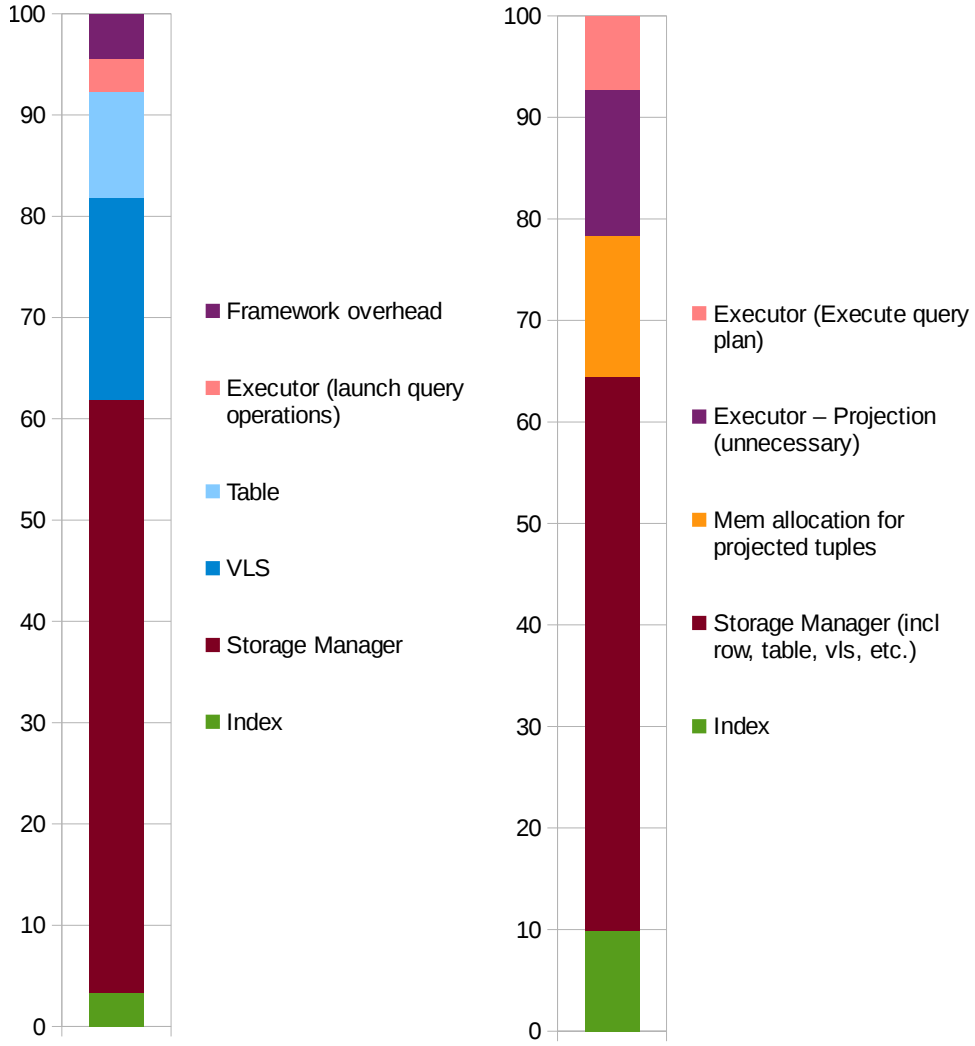

Fig. 12. Comparison of relative time breakdown for Slingshot, and PostgreSQL respectively, for a microbenchmark Reader transaction.

In jTPCC, we create 8 terminals sending transactions to PostgreSQL, and run the workload with 100% Payment transaction types. Slingshot's worker pool is also set to 8 threads, and each transaction performs the set of Payment queries.

As can be seen in Figure 11b), PostgreSQL has higher throughput than Slingshot, both for T-SM (1.7x) and T-LG (1.5x), due to the unnecessary serialization of writers in Slingshot. PostgreSQL's conflict detection allows transactions that access different rows to be executed concurrently and commit successfully. In Slingshot, the Transaction Manager module simply serializes transactions if there is *potential* for conflict at a column-level, even if conflicts would not in fact occur. Given that all TPC-C Payment transactions involve updates on all three tables, and touch the same columns in each of them, all transactions are completely serialized in Slingshot.

Although the scheduling policy in our sample Transaction Manager is far from ideal, Slingshot's throughput is not terribly lower than PostgreSQL, even with full serialization. When there are fewer conflicting transactions (as in the microbenchmark), Slingshot gets better throughput than PostgreSQL. We conclude that Slingshot can come close to an RDBMS engine, even in transactional scenarios where RDBMSs are supposed to excel. With the extra implementation effort to create a more sophisticated Transaction Manager, Slingshot may even outperform traditional relational engines on such workloads.

3) Breakdown by logical components - side-by-side comparison: The differences between PostgreSQL and Slingshot in terms of the time spent in various logical components are important to analyze for OLTP-type workloads as well. Unfortunately, profiling callgraphs for multiple concurrent transactions was not possible with valgrind, so we only conducted this analysis for the microbenchmark, using one transaction of each type (reader and writer). The analysis shown in Figures 12 and 13 raises a few interesting observations.

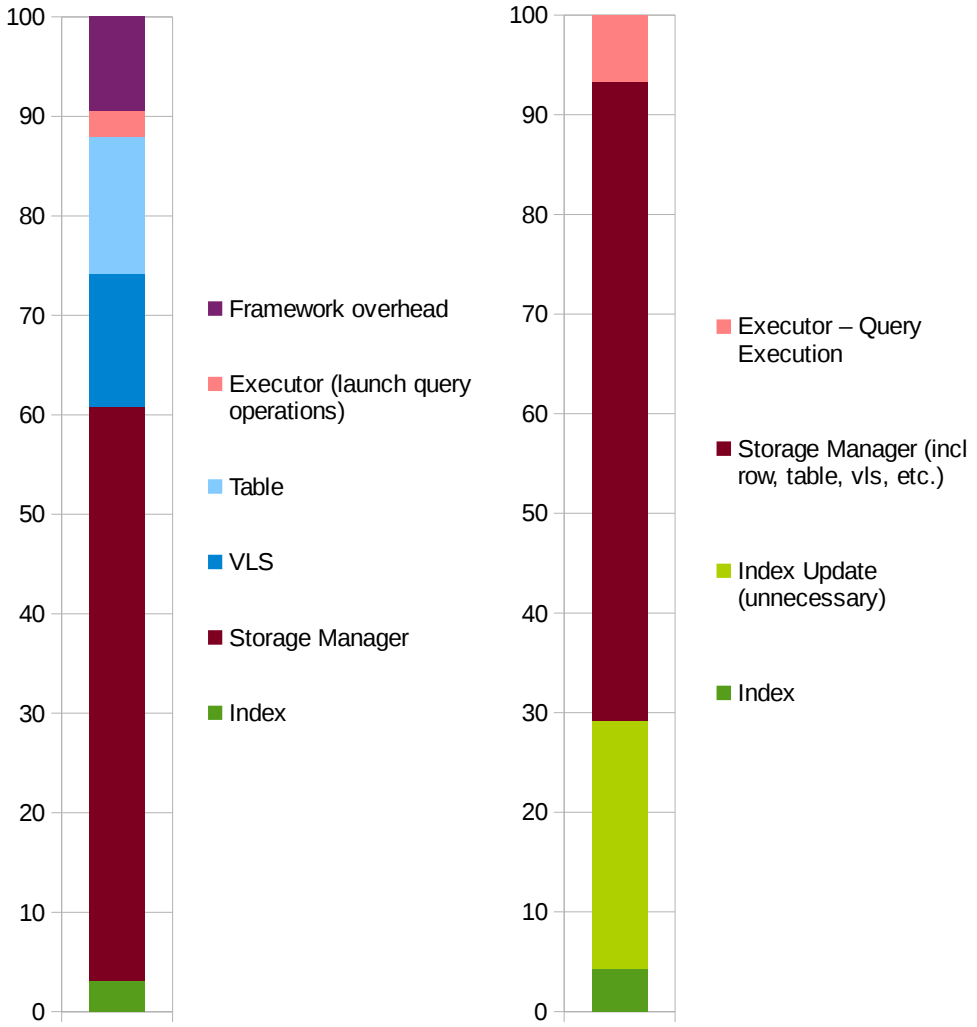

Fig. 13. Comparison of relative time breakdown for Slingshot, and PostgreSQL respectively, for a microbenchmark Writer transaction.

First, the Reader transaction in Slingshot also spends less relative time percentile in B-tree operations, due to two of the reasons pointed out previously (more entries per node, no support for variable index entries, and possibly more efficient locking). In PostgreSQL, the Reader transaction also spends a considerable amount of time doing a projection of the matching tuples, by the columns requested in the query. Slingshot uses a columnar projection in the data layout, thus eliminating the need for a projection altogether. Reconstructing the tuple is instead part of the Table and VLS costs, which seems to be a more efficient alternative in absolute value compared to PostgreSQL. Secondly, the Writer transaction incurs a cost of doing index updates, which is quite high and rather unnecessary. We should mention that the Writer transaction updates the Stock for the bought product, while the index used in the search involves the Category column. As a result, no index updates should occur at all. After deeper inspection, we noticed that whenever at least one field in a record gets updated, PostgreSQL's MVCC (multi-version concurrency control) mechanism creates a new record, and the older record is marked as invalid and garbage collected later. As a result, the corresponding indexes also have to be updated to reflect the change, even though the columns affected have not in fact been updated. In Slingshot, we do not incur such a problem due to the tailored column-oriented data layout, as well as the Storage Manager's versioning, whereby updates for fixed-size records are done in place (at the same offset, whether directly in the base version if no readers still refer to the older tuple, or in a new version in the versioning chain).

In all test cases presented so far, the side-by-side analysis highlights some of the ways in which Slingshot approaches custom design while providing generality to morph a new purpose engine. The modules implement the basic functionality which is necessary in any custom-solution, while the framework provides the generality to spawn a new purpose data processing engine.

VIII. Summary

Our work on optimizing the spatial filtering step by designing a new indexing framework was prompted by the observed limitations of traditional RDBMS engines (lack of modularity and design decisions that target generality which comes at a cost in performance). Consequently, we proposed a new data processing framework, which proved to be general enough to handle more than just spatial workloads.

In summary, Slingshot is a lightweight and modular data processing framework, designed to be flexible and highly customizable. Slingshot's approach resembles a microkernel operating system; it delineates database component functionality and provides a minimal core framework interconnecting them, while offloading functionality to external modules. As long as component interfaces are met, designers are free to decide what functionality or optimizations are suitable for their workload, without worrying about the impact on other components. In contrast, most RDBMS engines are strongly-coupled monolithic codebases, where relaxing constraints or dropping unnecessary functionality on a per-application basis is either not possible or a challenging task at best. Our results show that Slingshot achieves good performance by using distinct module implementations that are adapted to different workloads.

For the spatial workload tested, Slingshot outperforms PostgreSQL by an order of magnitude, and is 4 times faster than a custom solution (HadoopGIS). For the filtering step, going through Slingshot's internal component operations adds only an insignificant overhead compared to a custom stand-alone R-tree implementation.

IX. Integrating GPU parallel processing support in Slingshot

In a previous work [15], we have shown that GPU architectures show great potential as a platform to optimize a highly-parallel spatial task (refinement). We showcase the benefits of using GPUs as co-processors for speeding up the refinement step, when the parallelization is integrated in a full-fledged data processing engine.

While our efforts to integrate GPU support into PostgreSQL indicate that this is possible, the performance gains of offloading computations on a GPU are not very substantial. The reasons include post-filtering selectivity (which, when low, has a big impact on under-utilization of GPU cores), the limited flexibility in the processing paradigm (tuple-at-a-time), the need to have the data laid out in a way that is suitable to GPU processing, etc., which we will discuss next. Furthermore, even when speedup for refinement is reasonably high, the overall query speedup is severely limited by the filtering stage (which is sequential, much slower than in Slingshot, and can sometimes account for 50% of execution time). As a result, we proceed to integrate GPU support for refinement in Slingshot.

The goal is to determine how much of the performance benefits of GPUs can be leveraged in the Slingshot infrastructure and to find if there are limitations or trade-offs. For example, to make efficient use of GPU cores and memory bandwidth, the workload must involve parallel computations that keep busy a large portion of the processing units. As a result, the selectivity of the query (in terms of the number of candidates that make it to the refinement stage) is an important factor in making efficient use of the GPU resources (high number of cores and memory bandwidth). When faced with low post-filtering selectivity, offloading computation to the GPU may not be the best idea; in this case, the traditional tuple-at-a-time paradigm may need to be changed, to permit accummulating enough post-filtering candidates to properly utilize the GPU resources.

All in all, we wish to analyze what the limits of GPUs are in a database engine, or when faced with challenging workloads that may not make efficient use of GPU resources or their particular data processing mechanism.

A. Offloading post-filtering candidates on the GPU

We enhance the spatial functionality in Slingshot with a GPU extension, which consists of a set of kernels for spatial functionality, similar to those in [15], implementing the computational geometry algorithms like those from the GEOS library. Due to Slingshot's modularity, integrating GPU support for spatial refinement is much simpler compared to our initial attempts to do so in the PostgreSQL engine. Additionally, we do not incur the issues of working around the data structures from PostgreSQL or transforming its data layout in a form that is suitable for processing on the GPU (the data structures used to store shapes need to allow coalesced memory accesses, which are crucial for GPU performance).

For each outer shape in a join, the spatial refinement stage takes the post-filtering candidates and processes them individually to determine if the shapes overlap. We offload this functionality on the GPU, in order to evaluate in parallel the candidate set for each shape. To avoid repeated memory allocations on the GPU, we pre-allocate on the GPU an empty chunk of memory, dedicated for transferring in a considerable number of shapes. As such, for each outer shape, its candidate set is transferred into this pre-allocated chunk and the kernel corresponding to the spatial operator (e.g., intersects) is invoked.

The selectivity of the filtering stage is directly correlated with GPU utilization. To avoid GPU under-utilization, we must offload only large candidate sets that can keep busy a large number of cores and benefit from the high memory bandwidth. We implement a simple scheduling policy which uses the cardinality of the candidate set to determine which inner loops to offload to the GPU, and which ones to process on the CPU. The statistics of the query planner are used to determine the expected cardinality in this type of scheduling decision. We speculate that the ideal threshold above which GPUs are expected to be useful is somewhere around the number of its total cores. Nevertheless, we test this hypothesis to determine the optimal point.

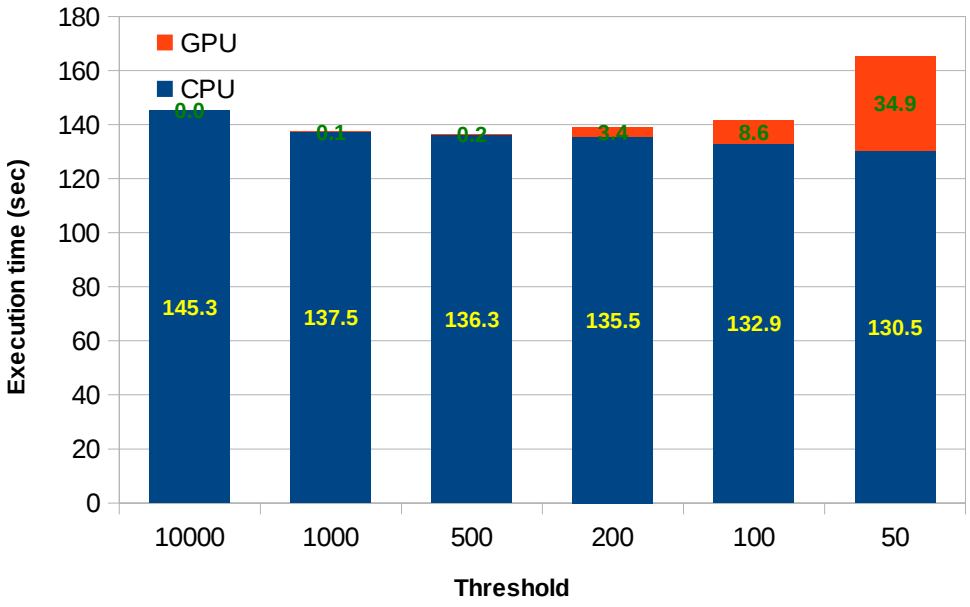

Fig. 14. Offloading candidate sets on the GPU

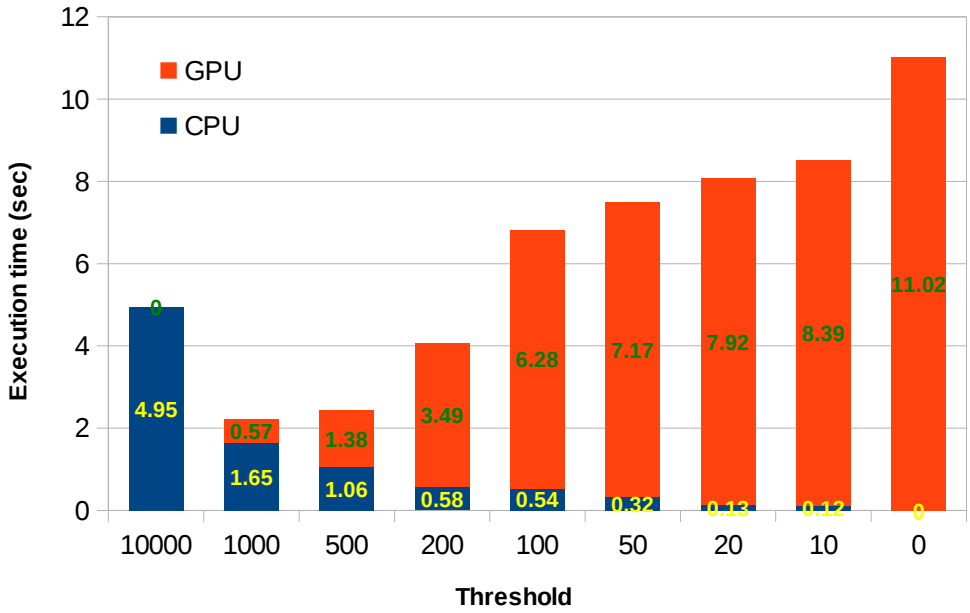

Fig. 15. Offloading candidate sets on the GPU for the a Join with higher post-filtering selectivity

For the long-running spatial join query described in Section VII-A, we vary the threshold for candidate cardinality and capture the time spent in refinement on the CPU (for the small candidate sets that fall below the threshold) and on the GPU (the large candidate sets above the threshold) in Figure 14. We use an NVIDIA GTX 580, with 512 cores, and 3GB global memory. Unfortunately, although the optimum point does seem to be around the number of cores on the GPU, we noticed there is no significant performance improvement by offloading refinement of candidate sets on the GPU. The reason is that for the dataset involved in this particular query, for most outer shapes the cardinality for their post-filtering candidate sets is very low, which leads to a severe underutilization of the GPU. The average cardinality of a candidate set (excluding empty ones) is roughly 6.4 shapes. Offloading a small number of candidate shapes on the GPU is highly inefficient since it spins up GPU cores that do not get used and does not fully benefit from the high memory bandwidth. With lower thresholds, the overall CPU + GPU time gets worse than processing everything on the CPU, because of low parallelism and the fact that GPU cores run at a much lower frequency than the CPU.

We experiment instead with the set of landmass polygons (arealm) from the TIGER Texas dataset, by performing the same

spatial intersection operation against the edges polylines from TIGER Texas. The spatial join between arealm and edges contains a larger percentage of candidate sets with high cardinality, which allows us to better observe the scheduling threshold. For this query, Figure 15 shows the fraction of time spent in computations done by the CPU and GPU. As can be seen, the optimal threshold is somewhere between 500 and 1000. An interesting observation is that using the GPU for candidate sets with low cardinality is again detrimental to overall execution time.

Overall, we note that the decision whether to schedule the refinement of candidate sets on the GPU is highly dependent on the dataset and the selectivity of the filtering step. Consequently, many queries may utilize the GPU suboptimally or may be better off processed on the CPU. In order to better take advantage of the data parallelism and leverage GPUs for spatial refinement more efficiently, we need to change the traditional tuple-at-a-time database processing paradigm.

B. Leveraging GPUs efficiently

DBMSs process relational operators in the query plan in a tuple-at-a-time fashion, where a tuple resulted from an operator (e.g., sequential scan) is 'pushed' upwards in the execution plan, *before* processing another tuple from the same operator. We change the processing model for our spatial join to use a set-at-a-time approach for refinement. Specifically, the filtering stage is performed for the outer shapes as before, except that the candidate sets the outer shapes are instead accummulated in batches, into a buffer, until a specific batch size is reached (see Figure 16). At this point, the accummulated chunk of candidate sets and their corresponding outer shapes are transferred on the GPU and the refinement kernels are launched. As a result, GPU refinement is now performed in multiple rounds (each round corresponding to a batch of candidates).

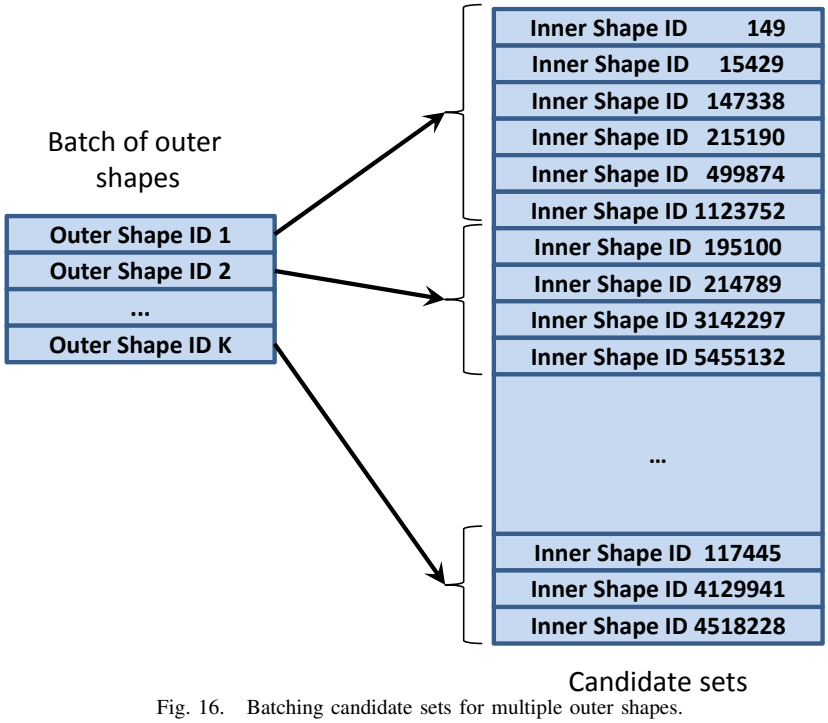

Fig. 16. Batching candidate sets for multiple outer shapes.

The cardinality of candidate sets in general has a high amount of skew, in the sense that the number of post-filtering matches for each outer shape can vary widely, which may in turn impact the refinement time. In order to ensure good load balancing among the GPU cores, we have to carefully pick the unit of computation for each GPU core. Designating each outer shape with its whole candidate set on one core, is far from ideal because in the presence of skew, the overall latency will be dictated by the largest candidate set.

Instead, we decided to assign each core an equal share of outer-inner shape pairs, in a round-robin fashion, taking into account the warp size specific to our GPU, as depicted in Figure 17. As a reminder, the basic unit of scheduling threads on a GPU is a *warp* (in NVIDIA terminology), or *wavefront* (for AMD hardware). Essentially, warps are a sub-division of work specific to the GPU, which allows the hardware to coalesce memory accesses and instruction dispatch.

Shapes are all appended in one large contiguous array of points (to simplify GPU transfers and pointer indirections), while metadata tracks where each shape starts and their sizes so that each core knows where to retrieve its assigned shapes. The batches of candidate sets solely consist of outer and inner pairs of shape IDs. For each pair of shapes, the corresponding thread will indicate whether the spatial predicate is met, and fill in the right location in the result set. The data structures involved in metadata bookkeeping are designed to efficiently leverage coalesced accesses to GPU memory. By storing only a minimal

amount of information we keep the memory footprint of metadata small. The metadata for each round is transferred in the pre-allocated large chunk of GPU global memory (as previously mentioned, we reserve a block of global GPU memory, to avoid repeated small CUDA memory allocations which are expensive).

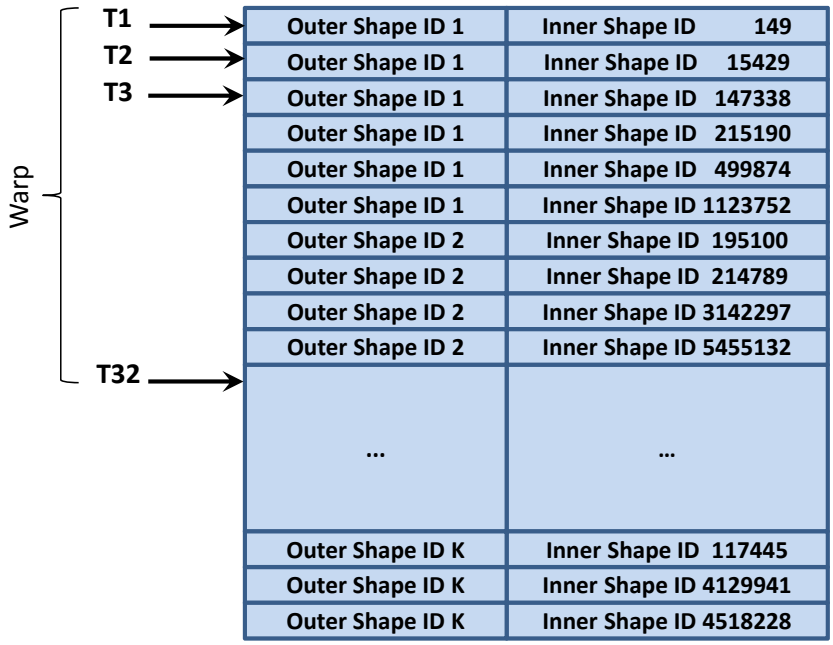

Fig. 17. Assigning candidate pairs round-robin to GPU context threads, modulo the thread warp size.

The skew problem manifests at the level of points within a shape as well. While we considered assigning computations for each core at the granularity of points or line segments, this turned out to be infeasible due to the bookkeeping complexity caused by two-level skew in the spatial dataset. Essentially, if the unit of computation is a line segment, we ran into a tradeoff with no good solution. On the one hand, we stored metadata for keeping track of where each line segment belongs, which ensures perfect load balancing and nearly-ideal memory coalescing, but fills up most of the (limited) GPU memory with metadata. This in turn means that the batch size is rather small and many cores sit idle, which leads to GPU under-utilization. On the other hand, we kept only shape "markers", which indicate where each shape starts and another ends, and based on shape size, could determine where a line segment belongs, to determine where to place the result in case of a match. Unfortunately, while this approach reduced the GPU memory footprint of metadata, and allowed for ideal load balancing, the memory accesses involved in determining the mapping of a line segment to a shape, caused poor memory access patterns. This was due to misaligned warps, and/or cores accessing the same data when processing different segments of the same shape. Consequently, we decided not to operate at this granularity.

C. Batch candidate set processing

Slingshot allows the flexibility to easily switch to batch-processing candidate sets, instead of the traditional tuple-at-a-time processing typical to DBMS operators. The execution plan is modified to allow batch processing with a predetermined batch size. The batch size is established based on statistics from the system Slingshot is running on: the amount of global GPU memory available (which determines how many candidates we can roughly accomodate, in conjunction with the histogram of shape sizes), number of GPU cores (to determine the number of shape pairs per core), CUDA capability and width of the GPU device system bus (which determines whether 32-bit reads, 64-bit reads, or even 128-bit reads may be coalesced).

We plug into Slingshot the spatial GPU processing kernels and GPU-side computational geometry engine as a separate module (more like a library of calls, rather than a module in the sense of Slingshot component implementations). We evaluate the Slingshot GPU-based refinement and compare it to the CPU-based equivalent. Figure 18 shows side-by-side the refinement time and total time for the spatial join for both versions of Slingshot (CPU-based and GPU-based). While the CPU-based refinement takes roughly 153 seconds (second bar), the GPU-based refinement is just under 15 seconds (third bar), for a total of 10.3x speedup.

With the filtering step being limited to running on the CPU, we compare the total execution time for the full spatial join (filtering + refinement), between the entirely CPU-based Slingshot version and the CPU+GPU hybrid version (the one that offloads refinement on the GPU). The hybrid version runs in 31.4 seconds, 5.4x faster than the CPU-based Slingshot (170 seconds).

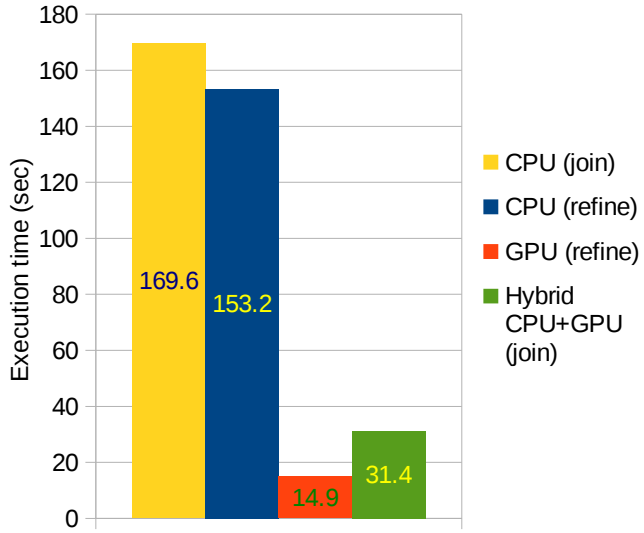

Fig. 18. Comparison of the Slingshot CPU-version and Slingshot using GPU-based refinement.

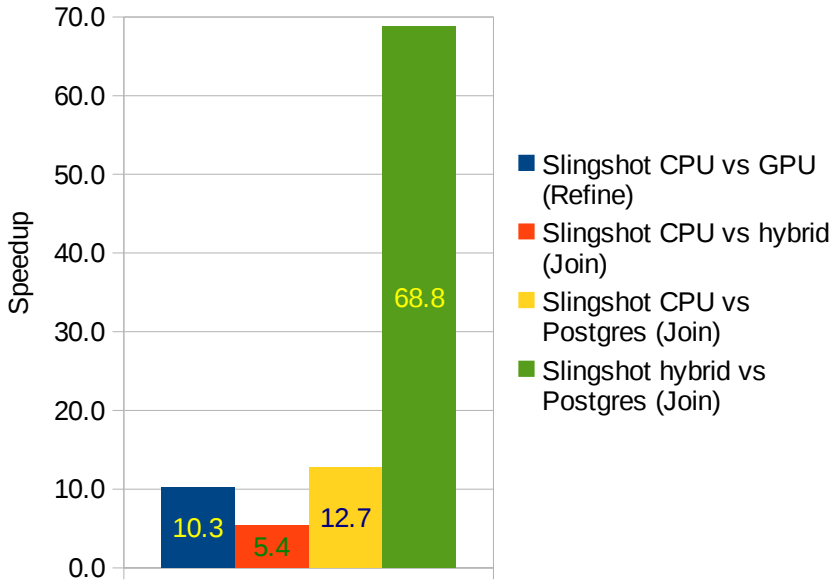

Fig. 19. Summary of sources of speedup for Slingshot; We show the speedup obtained by using GPUs for the refinement step, the total speedup achieved for the full join (including the CPU-based filtering step), and the speedup of both Slingshot versions compared to PostgreSQL on the long-running spatial analytics query.

In Figure 19, we summarize the speedups obtained by the hybrid Slingshot system compared to the CPU-based version, for both refinement and full-join. We also include the speedup of both Slingshot versions compared to PostgreSQL. The initial Slingshot version runs 12.7x faster than PostgresSQL, while the hybrid Slingshot runs close to 70x faster than PostgreSQL. Basically, we have shown that Slingshot can reduce a long-running spatial analytics query that would normally takes roughly 40 minutes, to just over half a minute. We conclude that not only does Slingshot enable more flexibility in design choices of either parts or all of the data processing engine, but it also allows users to employ new technology (such as GPU architectures) efficiently, in order to run faster data analytics.

X. Related work

The idea that no single database system is capable of simultaneously meeting both functionality and performance requirements of a wide range of applications has been around at least since the Exodus project in 1986 [17], which proposed a flexible system for building database applications based on a set of kernel DBMS facilities. Unfortunately, there was no single implementation of the entire system, but rather lots of separate pieces, leading to no concrete system for comparison.

Several decades later, database researchers noted that traditional RDBMS engines are losing ground to custom solutions designed specifically for a particular application domain [1], [2], [3]. For example, column-stores such as Vertica (based on C-Store [4]), MonetDB [5], ParAccel [6], and VectorWise [18], have been shown to outperform traditional row-stores on OLAP workloads. Other solutions take a hybrid approach. Plattner [19] discusses some common approaches to supporting both OLAP and OLTP. Grund et al. [20] propose Hyrise, a hybrid in-memory storage engine for OLAP and OLTP workloads. A different approach was taken by Kemper et al. [21] with HyPer, a hybrid main memory database that uses virtual machine snapshots to handle both OLAP and OLTP workloads simultaneously.

In contrast, Slingshot is not specifically designed for one purpose, nor is it a hybrid solution for OLTP and OLAP. Slingshot provides a minimal engine that only handles core functionality like the logical inter-operation of components. By exposing interfaces to DBMS components, Slingshot allows most design decisions to be made externally in the modules, in the most efficient manner for the target workload(s).

The conceptual ideas behind Slingshot are somewhat similar to those proposed by Irmert et al. [22], who discuss the potential of designing a modularized DBMS with interchangeable building blocks. However, their short paper did not provide much technical detail on how to implement such a system, lacked any evaluation, and no followup work ensued.

Modularity has been discussed as a desirable feature in the context of other genres of processing systems, at a completely different level than Slingshot. For example, in the context of Cloud storage, Cloudy [23] proposes a flexible distributed architecture that allows interchangeable schemes for data partitioning, routing model, and consistency protocol.

XI. Future directions

As future development work beyond the scope of this research work, we plan to extend Slingshot with a front-end to provide a traditional query parser and implement a dedicated planner module using the statistics exposed through the current interface, as well as adding more functionality and features (e.g., more operators like GroupBy, additional join types, more sophisticated modules such as adding compression and sorted columns to our ColumnStore, etc.). Since these features are captured within the role of a specific component, according to our three design principles, the core framework will remain lightweight and fast. Our interfaces are designed to support hooking in the extra features, thus ensuring that the core framework will not fall into the same bloating problem as many other RDBMS engines have before.

Another possible future direction is designing support for distributed query processing in Slingshot, and determining the effect of distributed data processing (replication, partitioning, etc.) on the abstractions and design principles proposed in Slingshot. Given the expected magnitude of this task, this may be in itself a stand-alone research topic that we plan to explore at a later time.

XII. Conclusion

Slingshot is a lightweight and modular data processing framework, designed to be flexible and highly customizable. Slingshot's approach resembles a microkernel operating system; it delineates database component functionality and provides a minimal core framework interconnecting them, while offloading functionality to external modules. Slingshot's components are abstracted into interfaces, which can be externally implemented as pluggable modules. As long as component interfaces are met, designers are free to decide what functionality or optimizations are suitable for their workload, and implement each component interface accordingly, without worrying about the impact on other components. In contrast, most RDBMS engines are strongly-coupled monolithic codebases, where relaxing constraints or dropping unnecessary functionality on a per-application basis is either not possible or a challenging task at best.

The advantage of our approach lies in the flexibility in selecting appropriate modules for the target data processing application. Module designers have the liberty to incorporate functionality as efficiently as possible (e.g., using optimized data structures, code that can leverage AVX extensions, etc.) without worrying about cross-component dependencies. Furthermore, this enables "shedding" functionality for performance, by allowing module implementations to omit features that are not necessary for the intended application.

The design decisions regarding how and where to draw the abstraction lines in Slingshot are inspired by some of the valuable lessons learned from prior efforts in achieving extensibility and customizability in current database engines. By enforcing the three design principles, we decoupled the logical database components into clear-cut interfaces and separated the role of each component from its implementation.

Slingshot achieves good performance by using distinct module implementations that are adapted to different workloads. The breakdown of work spent in each component clearly shows that the key to Slingshot's success lies in the fact that it treats useful work as a first-class citizen, while reducing or eliminating overheads caused by supporting unnecessary features for the job at hand. At the same time, Slingshot's configurability allows it to be morphed into a new engine targetted at a completely different workload. This demonstrates that our modular design for a data processing framework can provide a good balance between generality and performance.

We demonstrated not only that a flexible data processing framework need not sacrifice performance for generality, but also that Slingshot in fact enables easy integration of new technology that can speed up processing even further. By integrating GPU architectures, Slingshot can achieve overall performance improvements of roughly 70x over a traditional RDBMS on long-running spatial data analytics.

REFERENCES

[1] M. Stonebraker, C. Bear, U. Cetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik, "One size fits all? part 2: benchmarking results," in *CIDR*, 2007.

[2] M. Stonebraker and U. Cetintemel, ""One Size Fits All": An idea whose time has come and gone," in *ICDE*, 2005, pp. 2–11.

[3] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era: (it's time for a complete rewrite)," in *VLDB*, 2007, pp. 1150–1160.

[4] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A column-oriented DBMS," in *VLDB*, 2005, pp. 553–564.

[5] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *VLDB*, 1999, pp. 54–65.

[6] Y. Chen, R. L. Cole, W. J. McKenna, S. Perfilov, A. Sinha, and E. Szedenits, Jr., "Partial join order optimization in the ParAccel analytic DB," in *SIGMOD*, 2009, pp. 905–908.

[7] B. Simion, D. N. Ilha, A. D. Brown, and R. Johnson, "The price of generality in spatial indexing," in *BigSpatial*, 2013, pp. 8–12.

[8] M. A. Olson, K. Bostic, and M. Seltzer, "Berkeley DB," in *USENIX ATC*, 1999, pp. 43–43.

[9] "MySQL," http://www.mysql.com.

[10] "PostgreSQL," http://www.postgresql.org/.

[11] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: How different are they really?" in *SIGMOD*, 2008, pp. 967–980.

[12] S. Ray, B. Simion, and A. D. Brown, "Jackpine: A benchmark to evaluate spatial database performance," in *Intl. Conf. on Data Engineering*, April 2011.

[13] "TIGER®, TIGER/Line® and TIGER®-Related Products," http://www.census.gov/geo/www/tiger.

[14] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop GIS: A high performance spatial data warehousing system over MapReduce," *VLDB Endow.*, vol. 6, no. 11, pp. 1009–1020, Aug. 2013.

[15] B. Simion, S. Ray, and A. D. Brown, "Speeding up spatial database query execution using gpus," in *Proceedings of the International Conference on Computational Science, ICCS 2012, Omaha, Nebraska, USA, 4-6 June, 2012*, 2012, pp. 1870–1879. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2012.04.205

[16] "jTPCC - open source Java implementation of the TPC-C benchmark," http://jtpcc.sourceforge.net.

[17] M. J. Carey, D. J. Dewitt, D. Frank, G. Graefe, J. E. Richardson, E. J. Shekita, and M. Muralikrishna, "The architecture of the EXODUS extensible DBMS," in *OODS workshop*, 1986, pp. 52–65.

[18] M. Zukowski and P. Boncz, "From X100 to Vectorwise: Opportunities, challenges and things most researchers do not think about," in *SIGMOD*, 2012, pp. 861–862.

[19] H. Plattner, "A common database approach for OLTP and OLAP using an in-memory column database," in *SIGMOD*, 2009, pp. 1–2.

[20] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, "HYRISE: A main memory hybrid storage engine," *Proc. VLDB Endow.*, vol. 4, no. 2, pp. 105–116, Nov. 2010.

[21] A. Kemper and T. Neumann, "Hyper: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots," in *ICDE*, 2011, pp. 195–206.

[22] F. Irmert, M. Daum, and K. Meyer-Wegener, "A new approach to modular database systems," in *SETMDM workshop*, 2008, pp. 40–44.

[23] D. Kossmann, T. Kraska, S. Loesing, S. Merkli, R. Mittal, and F. Pfaffhauser, "Cloudy: A modular cloud storage system," *VLDB*, vol. 3, no. 1-2, pp. 1533–1536, Sep. 2010.