1 What is Backstab?

Backstab is a construction set for building a custom database servers in Objective Caml programming language, which is well–situated for such a tasks. The primary purpose is to reimplement an old `BDMS`/Spires database management system used in particle physics, but you can build any other database server architecture using the most common server–side features like internal request queues and forwarders, threads pool, low–level data file format (datablock pools (e.g. for tables representation), custom indexes (sorted lists, B–trees, trees, hash tabs, etc.), LOB–pools, pure functional structures pools) and simple network wrappers.

It is in a pre–alpha stage now, but development process is very fast (thanks to OCaml team for a great language!), and you can already download some components to play with.

1.1 History

There was BDMS — a pile of Fortran code used to produce a standalone non–relational database applications. And there is a lot of very urgent data stored under BDMS — all the Particle Physics Data System managed by COMPAS Group at Institute for High Energy Physics. BDMS is huge, slow, poor in a basic functionality and a very nonportable (it works now under VMS only), so, we decided to reimplement it completely from scratch. Sure there was some attempts to move all data into relational structures, but we failed. The result was a read–only “database management system” with an embedded Lisp interpreter which was able to read BDMS exports — LDMS project. I used it to build a WWW–interface to PPDS data, and then I realized a power of Lisp (and functional programming in general) in a database area. The first version of LDMS was designed very poorly, with read–only access in mind, so I failed in attempts to create a full–featured DBMS–server from it. I used to give up that project and started from scratch again, now almost completely in a functional language to prove it’s power. I was working for Particle Data Group at LBNL, and there was not much time for this project, but I managed to build a basis for a feature work. Now we have it, now it’s time to improve it.

2 Architecture

The whole engine is separated into the following levels:

- *low level*: Storage and networking (messaging) features. Storage is a fixed–length blocks file with atomic blocks allocation and deletion, chains of blocks and a fast cache for blocks and chains metadata. There are also some “special blocks” features, like “directory” chains. Networking is a simple event–based packet system. The very low level is a special datatypes — Row and Serialized, which is used for all operations with blocksfiles.

- *server engine*: This layer contains threads pool, request queues, higher level networking functions, some most generic data formats over blocksfiles (e.g. configuration info chains, generic indexes chains, data record pools, raw binary files, functional objects pools). Almost all features of this layer developed on a higher level (reference database engine implementation), and then, after a proven feature generalization it will be moved on this common level.

- *DBMS server implementation*: Here will be all user–defined specific data formats and networking protocols, request language interpretation, startup, shutdown and configuration sequences, and all other DBMS stuff. All generic features from this level, which may be useful for other DBMS implementations must be moved to *server engine* layer.

Low level features are all coded in a C language, ’cause we will need them in a client libraries and tools, which may be coded in some other languages then OCaml. For Java we will need a separate library, completely rewritten from scratch. But all higher levels of server engine are programmed in OCaml — it’s just a prototyping system and playground, so we don’t need a super–performance. Btw., OCaml performance is good enough even for a production heavy–duty release, so we can optimize it all later.

Please note that by default this library is thread–safe, but if you don’t want to use threads you can disable ’em with `-DNOTHREADS` option. Backstab is developed multithreaded, but with a single–threaded variant in mind (e.g. for embedded applications and for some stupid platforms which does not support POSIX threads).

3 Low level layer architecture

Okee, let's go into some details — you have to understand an overall architecture before reading sourcecode documentation.

3.1 Datatypes and containers

There are some basic modules in a low level that implement some specific data types operations and representation, and some generic memory containers with a debugging features.

3.1.1 Integer–key containers

The module `blocks_containers` implements a simple integer–key container. It's main purpose is to keep an objects of a same type together, with integer references, not pointers, with a possibility to dump or finalize the whole it's contents. This module provides an interface for a container creation and deletion, object allocation and finalization (with a mandatory finalization function). For a details please read CWEB sourcecode documentation for this module.

3.1.2 Row datatype

It's a "row" of raw data — a kind of growing bytearray. You can not access it directly (sure, you CAN, but it will be a dirty hack) — the module `blocks_row` provides an interface only for some specific operations. Details are in the CWEB documentation.

3.1.3 Serialized datatype

Really it's not a "serialized", but a "serializeble" type. The real serialized type is a Row, which can be embedded in a network packet, saved in a file, etc. "Serialized" is a one–way list of elements, which can be "byte", "16–bit word", "32–bit word", "64–bit word", "ASCIIZ–string", "String" or "Row". It is used as a temporary storage type in internal transmission — e.g., you can read Serialized directly from blocksfile or a Row using a simple format specification like `'bb3s[10]b(5)'` (byte+byte+32bit+string(10 chars max)+5 bytes). May be later more expressive format specifications will be implemented (or, as an alternative, format callback declarations, which will be more functional and more fun).

3.1.4 Hash tables

It's an internal storage which can't be used on a higher levels. Hash tables are implemented in modules **`ihash`**, **`ahash`**, **`aihash`** — integer–key table, ASCII–key table and ASCII–key table contains integers. It's just a quick hack, not as fast as hash tables can be, and may be even leaky — so be careful with 'em.

3.2 Debugging features

Low level uses `malloc()` feature for some allocations outside Containers. And, sure, there can be some stupid memory leaks. To catch 'em we're using "paranoid malloc debugger" from module `paranoid`. Just add `-DPARANOID` to a C compiler options to enable memory debugging.

Some other debugging features can be enabled with a `-DLLDEBUG` option — it's a logging output, structure types assertions, etc.

3.3 Event–based network packets

It's a very simple interface to sockets, which works as follows:

1. Server registers a listener (one listener per thread). Every listener have a callback which is used to register a packet callback for every new connection established.

2. When a new connection is created, listener callback registers a new callback (OCaml closure, for example) to process arriving packets. Every incoming packet represented as a Row, so callback knows its length.

3. Packet callback is executed in a listener thread context, so, it have to be small and fast. The common way is to parse incoming packet into the internal Request and then pass it to the request queue, which runs in it's own thread. In a one–thread environment execution of listener and request queue must be sequental — one `select()`, then one request, and so on.

4. `pkt_send()` function is executed in a caller thread. May be later it will only pass a new packet to an internal queue and then it will be processed in a listener context. I do not know yet which way is better, 'cause we consider a packet sending process as very fast and not resource–consuming.

This interface implementation is in `packets` module. OCaml bindings are in the `pkts.ml` module. Please check out CWEB and OCamlWEB documentation for details.

You can disable this module with `-DNO_NETSTUBS` option.

3.4 Blocksfile

Blocksfile is an interface to a file of fixed–length blocks which can be packed in a chains (something like regular files). You can allocate a block or a chain of blocks, read a chain, store some data in a chain, read data from a chain, change a chain length, remove a chain, store a chain position in a "directory", and so on. It is a basis for all data storage features of a database management server.

4 OCaml interface for a low level

This interface is very simple. It's defined in two modules — `Blocksfile` and `Pkts`, so, please refer to that implementations for a details. Some more details can be found in a `blockstubs` module.

4.1 Data types

The following special datatypes defined: `bfile`, `blocknum`, `datachain`, `b_byte`, `b_u16`, `b_u32`, `b_row`, `b_serialized`, `b_ser_value`, `np_connection`, `np_listener`. That's enough to describe all our objects. The main type you'll work with is a `b_ser_value` which embedds all other `b_*` types. There are simple functions to convert Serialized to `b_ser_value list` and vice versa, and to dump Serialized into Row. Please remember that you have to convert OCaml integers to our special types with appropriate functions, and remember that our integers are unsigned and may not fit in OCaml signed 32–bit integer.

4.2 Common usage

5 Generic server layer architecture

As I already mentioned, server layer is a library of common tools which may be useful in any database server engine architecture. Some of them are specific for my own ideas, some are very common.

5.1 Context, Server and Request

We're using a very specific terminology here: we will call a special function which is executed in a separate thread, triggered by a "request" as a "server". Every "server" have it's "context", which includes its mutex and some server–specific info. Again — server does not own a thread. It's assigned to a free thread from pool when internal request to this server comes to the queue head and the server context mutex is free, and after its execution it releases a thread to the pool again. In a one–thread environment servers are just executed from a queue sequentally.

5.2 Request queue and special Servers

Request queue is a main bus of any server engine. Incoming client requests are executed by special listener servers and splitted into some internal atomic requests (e.g. "select" something from index, produce a set of record numbers, cache it somewhere, fetch some of that records from record pool and send them back to client), and that requests goes into that main queue to be executed by a servers. Every request have "source" and "destination" — IDs of server contexts of its source and destination, destination server instruction code and a list of args if instruction needs them. Request can be a "simple request", "response" or "error". Some requests can be serialized and forwarded via socket interface to other process, if you want to build a distributed engine architecture. We have a default forwarder server already, if you don't need any special functionality like load balancing.

5.3 "socket"–based request handling

Any server can have a conversation with multiple servers, so it have to know multiple contexts simultaneously. We're using "sockets" here — it's a integer–key hashtable with actual request–processing closures, and a context key is created in a begining of conversation. So, if we're waiting a response from server A to continue some our work, and at the same time we can receive a completely new instruction from listener server and a response from other server B, we must keep 3 contexts — default context (0) for a new requests and two for an old ones. Socket key is a part of request.

6 BDMS–like reference architecture

The main goal of the whole project is to build a full featured non–relational scalable database server. Why non–relational? We already have a damn lot of relational database servers, we don't need another one *SQL. But there is a lack of a new ideas in database area, and good old ideas are just forgotten. It's strictly proven that relational model is enough (sure, it's just a Cartesian product with types, any functional programmer have to know that it's enough for representation of any data structure you can ever imagine). But it doesn't mean that it's effective for all types of structures. The most common application of databases is a *document* storage. And documents are all hierarchial structured. We all know XML as a damn good representation for structures, relations, documents - so, we need a database with a same structure for it's records.

Like an XML, we have to split document into structure specification (DTD) and raw data (XML). Following old BDMS terminology we will call them FDT (File Definition Table) and Records. FDT declares a table structure as a hierarchy of Data Elements. Check out the following example of such a structure:

- `DOC_ID:number:mandatory:key` — Unique key for this document
- `DOC_TITLE:string:key` — Title for this document, not mandatory
- `AUTHOR.:node` — Author info node, can have multiple occurences, which is specified by the dot in Data Element name.
  - `NAME:string:key:mandatory` — Author name
  - `ID:number:key` — ID to link with other table, if this author is mentioned there. Not mandatory.
  - `INST.:string:key` — Institution or collaboration where this author comes from. He can take a part in a multiple collaborations, so we place a dot in a name here.

- `ABSTRACT:text` — big (multiline) text, article abstract.
- `TOPIC.:string:key` — keywords for this article.
- ... and so on...

As you can see, we have to use two tables linked with a key to build the same structure in a relational model. And this is a simplest example. The data I work with fits in one BDMS table, but it needs more then 100 relational tables and a lot of SQL code to produce a simple report from a simple request. For this kind of data this structure will be much more effective then relational model, and this is very urgent for online WWW applications with a lot of users.

6.1 Servers

This implementation is build up from the following special servers:

- Record Pool — manages a container for a packed record data. It's job is to cache record data, manage free space, collect user–defined filters, and so on.
- Index — it's a manager for key data elements. It can look into the index using specified predicate and then produce a Set of record numbers and some specific info (version numbers, etc.).
- Set — sets produced by requests and logic can be very huge, so we need a special server to handle 'em.
- Connection — this one talks to client application and interpretes a request language.
- Table — just a wrapper which holds all info about objects belongs to the table — IDs of Record Pool and Index servers, etc.
- FDT — manages a FDT attached to the table.

Some of them will go into "server engine" layer later.