FastDB Main Memory Database Management System

Konstantin Knizhnik

March 22, 1999

Contents

List of Tables

1 Introduction

FastDB is highly efficient main memory database system with realtime capabilities and convenient C++ interface. FastDb doesn't support client-server architecture and all application using FastDB database should run at the same host. FastDB is optimized for applications with dominated read access pattern. High speed of query execution is provided by elimination of data transfer overhead and very effective locking implementation. Database file is mapped to virtual memory space of each application working with database. So query is executed in the context of application, requiring no context switching and data transfer. Synchronization of concurrent database access is implemented in FastDB by means of atomic instructions, adding almost no overhead to query processing. FastDB assumes that all database is present in RAM and optimizes search algorithms and structures according to this assumption. Moreover FastDB has no overhead of database buffers management and need not to transfer data between database file and buffers pool. That is why FastDB will work significantly faster then traditional database with all data cached in buffers pool.

FastDB supports transactions, online backups and automatic recovery after system crash. Transaction commit protocol is based on shadow root pages algorithm, performing atomic update of database. Recovery can be done very fast, providing high availability for critical applications. Moreover, elimination of transaction logs improves total system performance and leads to more effective usage of system resources.

FastDB is application-oriented database. Database tables are constructed using information about application classes. FastDB supports automatic scheme evaluation, allowing you to do changes only in one place — in your application classes. FastDB provides flexible and convenient interface for retrieving data from database. SQL-like query language is used to specify queries, and such post-relational capabilities as non-atomic fields, nested arrays, user-defined types and methods, direct interobject references simplifies design of database application and makes them more efficient.

Although FastDB is optimized in the assumption that all database fits in computer physical memory, it is also possible to use it with databases, which size exceeds size of physical memory in the system. In the last case standard operation system swapping mechanism will work. But all FastDB search algorithms and structures are optimized in the assumption of residence of all data in memory, so they efficiency for swapped out data will not be very high.

2 Query language

FastDB supports query language with SQL-like syntax. FastDB uses notation more popular for object-oriented programming then for relational database. Table rows are considered as object instances and the table — as class of these objects. Unlike SQL, FastDB is oriented on work with objects instead of SQL tuples. So the result of each query execution is a set of objects of one class. The main differences of FastDb query language from standard SQL are:

1. There are no joins of several tables and nested subqueries. Query always returns set of objects from one table.

2. Standard C types are used for atomic table columns.

3. There are no NULL values, except null references. I am completely agree with C. J. Date critics of three-value logic and his proposal to use default values instead.

4. Structures and arrays can be used as record components. Special exists quantor is provided for locating element in arrays.
5. User methods can be defined for table records (objects) as well as for record components.
6. User functions with single string or numeric argument can be defined by application.
7. References between objects are supported including automatic support of inverse references.
8. Construction start from follow by performs recursive records traversal using references.
9. As far as query language is deeply integrated with C++ classes, case sensitive mode is used for language identifiers as well as for keywords.
10. No implicit conversion of integer and floating types is done to string representation. If such conversion is need, it should be done explicitly.

The following rules in BNF-like notation specifies grammar of FastDB query language search predicate:

Example	**Meaning**
expression	non-terminals
not	terminals
|	disjoint alternatives
(**not**)	optional part
1..9	repeat zero or more times

Table 1: Grammar conventions

select_condition ::= (*expression*) (*traverse*) (*order*)
expression ::= *disjunction*
disjunction ::= *conjunction*
| *conjunction* **or** *disjunction*
conjunction ::= *comparison*
| *comparison* **and** *conjunction*
comparison ::= *operand* = *operand*
| *operand* != *operand*
| *operand* <> *operand*
| *operand* < *operand*
| *operand* <= *operand*
| *operand* > *operand*
| *operand* >= *operand*
| *operand* (**not**) **like** *operand*
| *operand* (**not**) **like** *operand* **escape** *string*
| *operand* (**not**) **in** *operand*
| *operand* (**not**) **in** *expressions_list*
| *operand* (**not**) *between operand* **and** *operand*
| *operand is* (**not**) *null*
operand ::= *addition*
additions ::= *multiplication*
| *addition* + *multiplication*
| *addition* || *multiplication*
| *addition* − *multiplication*
multiplication ::= *power*
| *multiplication* ∗ *power*
| *multiplication* / *power*
power ::= *term*
| *term*ˆ*power*
term ::= identifier | number | string
| **true** | **false** | **null**
| **current** | **first** | **last**
| (*expression*)
| **not** *comparison*
| − *term*
| *term* [*expression*]
| *identifier* . *term*
| *function term*
| **exists** *identifier* : *term*
function ::= **abs** | **length** | **lower** | **upper**
| **integer** | **real** | **string**
| **sin** | **cos** | **tan** |**asin** | **acos**
| **atan** | **log** | **exp** | **ceil** | **floor**
string ::= ′ { {*any_character_except_quote* } (″) } ′
expressions_list ::= (*expression* { , *expression* })
order ::= *order by sort_list*
sort_list ::= *field_order* { ,*field_order* }
field_order ::= *field* (*asc* | *desc*)
field ::= *identifier* { . *identifier* }
fields_list ::= *field* {, *field*}
user_function ::= *identifier*

Identifiers are case sensitive, begin with `a..z`, `A..Z`, `'_'` or `'$'` character, contain only `a-z`, `A..Z`, `0..9`, `'_'` or `'$'` characters, and do not duplicate a SQL reserved words.

`abs`	`acos`	`and`	`asc`	`asin`
`atan`	`between`	`by`	`cos`	`ceil`
`current`	`desc`	`escape`	`exists`	`exp`
`false`	`floor`	`in`	`integer`	`is`
`length`	`like`	`log`	`lower`	`not`
`null`	`or`	`real`	`sin`	`string`
`tan`	`true`	`upper`		

Table 2: List of reserved words

ANSI-standard comments may also be used. All character from double-hyphen to the end of the line are ignored.

FastDB extends ANSI standard SQL operations by supporting bit manipulation operations. Operators `and/or` can be applied not only to boolean operands but also to operands of integer type. Result of applying and/or operator to integer operands is integer value with bits set by bit-`AND`/bit-`OR` operation. Bitwise operations can be used for efficient implementation of small sets. Also rasing to a power operation ˆ is supported by FastDB for integer and floating point types.

2.1 Structures

FastDB accepts structures as components of records. Field of the structure can be accessed using standard dot notation: `company.address.city`

Structure fields can be indexed and used in order by specification. Structures can contain other structures as their components and there are no limitations on nesting level.

Programmer can define methods for structures, which can be used in queries with the same syntax as normal structure components. Such methods should have no arguments except pointer to the object to which they belong (this pointer in C++), and should return atomic value (of boolean, numeric, string or reference type). Also method should not change object instance (immutable method). If method returns string, then this string should be allocated using new char operator, because it will be deleted after copying of its value. So user-defined methods can be used for creation virtual components — components which are not stored in database, but instead if this are calculated using values of other components. For example, FastDb `dbDateTime` type contains only integer timestamp component and such methods as `dbDateTime::year()`, `dbDateTime::month()`,... So it is possible to specify queries like: `"delivery.year = 1999"` in application, where delivery record field has `dbDateTime` type. Methods are executed in the context of application, where they are defined, and are not available to other applications and interactive SQL

2.2 Arrays

FastDB accepts arrays with dynamic length as components of records. Multidimensional arrays are not supported, but it is possible to define array of arrays. FastDB provides a set of special constructions for dealing with arrays:

1. It is possible to get the number of elements in the array by `length()` function.
2. Array elements can be fetched by `[ ]` operator. If index expression is out of array range, then exception will be raised.
3. Operator in can be used for checking if array contains value specified by left operand. This operation can be used only for arrays of atomic types: with boolean, numeric, reference or string components.

4. Iteration through array elements is performed by `exists` operator. Variable specified after `exists` keyword can be used as index in arrays in the expression preceded by `exists` quantor. This index variable will iterate through all possible array index values, until value of expression will become true or index runs out of range. Condition

```
exists i: (contract[i].company.location = 'US')
```

will select all details which are shipped by companies located in US, while query

```
not exists i: (contract[i].company.location = 'US')
```

will select all details which are shipped only from companies outside US.

Nested exists clauses are allowed. Using nested `exists` quantors is equivalent to nested loops using correspondent index variables. For example query

```
exists colon: (exists row: (matrix[colon][row] = 0))
```

will select all records, containing 0 in elements of matrix field, which has type array of array of integer. This construction is equivalent to the following two nested loops:

```
bool result = false;
for (int colon = 0; colon < matrix.length(); colon++) {
    for (int row = 0; row < matrix[colon].length(); row++) {
        if (matrix[colon][row] == 0) {
            result = true;
            break;
        }
    }
}
```

Order of using indices is significant! Result of the following query execution

```
exists row: (exists colon: (matrix[colon][row] = 0))
```

will be completely different with result of previous query. The program can simply hang in last case due to infinite loop for empty matrices.

2.3 Strings

All strings in FastDB have varying length and programmer should not worry about specification of maximal length for character fields. All operations acceptable for arrays are also applicable to strings. In addition to them strings have a set their own operations. First of all string can be compared with each other using standard relation operators. At the current moment FastDB supports only ASCII character set (corresponds to type char in C) and byte-by-byte comparison of strings ignoring locality settings.

Construction like can be used for matching string with a pattern containing special wildcard characters `'%'` and `'_'`. Character `'_'` matches any single character, while character `'%'` matches any number of characters (including 0). Extended form of `like` operator with `escape` part can be used to handle characters `'%'` and `'_'` in the pattern as normal characters if they are preceded by special escape character, specified after `escape` keyword.

It is possible to search substring within string by `in` operator. Expression `('blue' in color)` will be true for all records which `color` fields contains `'blue'` word. If length of searched string is greater than some threshold value (currently 512), then Boyer-Moore substring search algorithm is used instead of straightforward search implementation.

Strings can be concatenated by `+` or || operators. Last one was added only for compatibility with ANSI SQL standard. As far as FastDB doesn't support implicit conversion to string type in expressions, semantic of operator + can be redefined for strings.

2.4 References

References can be dereferenced using the same dot notation as used for accessing structure components. For example the following query

```
company.address.city = 'Chicago'
```

will access record referenced by company component of `Contract` record and extract `city` component of `address` field of referenced record from `Supplier` table.

References can be checked for `null` by `is null` or `is not null` predicates. Also references can be compared for equality with each other as well as with special `null` keyword. When `null` reference is dereferenced, exception is be raised by FastDB.

There is special keyword `current`, which can be used to get reference to current record during table search. Usually `current` keyword is used for comparison of current record identifier with other references or locating it within array of references. For example, the following query will search in `Contract` table for all active contracts (assuming that field `canceledContracts` has `dbArray< dbReference<Contract> >` type):

```
current not in supplier.canceledContracts
```

FastDB provides special construction for recursive traverse of records by references:

`start from root-references`

`{ follow by` *list_of_reference_fields* `}`

First part of this construction is used to specify root objects. Nonterminal root-references should be variable of reference or array of reference type. Two special keyword `first` and `last` can be used here, locating first/last record in the table correspondingly. If you want to check for some condition all records referenced by array of references or single reference field, then this construction can be used without follow by part.

If you specify `follow by` part, then FastDB will recursively traverse table records starting from root references and using list of reference fields *list_of_reference_fields* for transition between records. *list_of_reference_fields* should consist of fields of reference or array of reference type. Traverse is done in depth first top-left-right order (first we visit parent node and then siblings in left-to-right order). Recursion is terminated when null reference is accessed or already visited record is referenced. For example the following query will search tree records with weight large than 1 in TLR order:

```
"weight > 1 start from first follow by left, right"
```

For the following tree:

			A:1.1			
	B:2.0				C:1.5	
D:1.3		E:1.8		F:1.2		G:0.8

result of query execution will be:

```
('A', 1.1), ('B', 2.0), ('D', 1.3), ('E', 1.8), ('C', 1.5), ('F', 1.2)
```

2.5 Functions

FastDB application can define its own functions. Function should have single argument of `int8`, `real8` or `char const*` type and return value of `bool`, `int8`, `real8` or `char*` type. User functions should be registered by `USER_FUNC(f)` macro, which creates static object of `dbUserFunction` class, binding the function pointer and the function name. For example the following statements makes it possible to use `sin` function in SQL statements:

Name	Argument type	Return type	Description
abs	integer	integer	absolute value of the argument
abs	real	real	absolute value of the argument
integer	real	integer	conversion of real to integer
length	array	integer	number of elements in array
lower	string	string	lowercase string
real	integer	real	conversion of integer to real
string	integer	string	conversion of integer to string
string	real	string	conversion of real to string
upper	string	string	uppercase string

Table 3: Predefined functions

```
#include <math.h>
...
USER_FUNC(sin);
```

Functions can be used only within application, where they are defined. Functions are not accessible from other applications and interactive SQL. Function returning `string` type should allocate returned value by operator `new`, because FastDB will call destructor after copying returned value.

In FastDB function argument can be (but not necessarily) enclosed in parentheses. So both of the following expressions are valid:

```
'$' + string(abs(x))
length string y
```

3 C++ interface

One of the primary goals of FastDB is to provide flexible and convenient application language interface. Anyone who have to use ODBC or similar SQL interfaces will understand what I am speaking about. In FastDB query can be written in C++ in the following way:

```
dbQuery q;
dbCursor<Contract> contracts;
dbCursor<Supplier> suppliers;
int price, quantity;
q = "(price >=",price,"or quantity >=",quantity,
    ") and delivery.year=1999";
// input price and quantity values
if (contracts.select(q) != 0) {
    do {
        printf("%s\n", suppliers.at(contracts->supplier)->company);
    } while (contracts.next());
}
```

3.1 Table

Data in FastDB is stored in tables which corresponds to C++ classes and class instances — to table records. The following C++ types are accepted as FastDB record atomic components:

Type	Description
`bool`	boolean type (`true`, `false`)
`int1`	one byte signed integer (`-128..127`)
`int2`	two bytes signed integer (`-65536..65536`)
`int4`	four bytes signed integer (`-2147483647..2147483647`)
`int8`	eight bytes signed integer ($-2^{63}..2^{63}-1$)
`real4`	four bytes ANSI floating point type
`real8`	eight bytes ANSI double precision floating point type
`char const*`	zero terminated string
`dbReference<T>`	reference to class `T`
`dbArray<T>`	dynamic array of elements of type `T`

In addition to types specified in the table above, FastDB records can also contain nested structures of these components. FastDB doesn't support unsigned types to simplify query language, eliminate bugs caused by signed/unsigned comparison and reduce size of database engine.

Unfortunately C++ provides no way to get metainformation about a class at runtime (RTTI is not supported by all compilers and also doesn't provide enough information). That is why programmer has to explicitly enumerate class fields to be included in database table (it also makes mapping between classes and tables more flexible). FastDB provides a set of macros and classes to make such mapping as simple as possible.

Each C++ class or structure, which will be used in database, should contain special method describing its fields. Macro `TYPE_DESCRIPTOR(field_list)` will construct this method. The single argument of this macro is enclosed in parentheses list of class fields descriptors. If you want to define some methods for the class and make them available for database, then macro `CLASS_DESCRIPTOR(name, field_list)` should be used instead of `TYPE_DESCRIPTOR`. Class name is needed to get references to member functions.

The following macros can be used for construction field descriptors:

`FIELD(name)`

Non-indexed field with specified name.

`KEY(name, index_type)`

Indexed field. `index_type` should be a combination of `HASHED` and `INDEXED` flags. When `HASHED` flag is specified, FastDB will create a hash table for the table using this field as a key. When `INDEXED` flag is specified, FastDB will create a T-tree (special kind of index) for the table using this field as a key.

`SUPERCLASS(name)`

Specifies information about base class (parent) of the current class.

`RELATION(reference, inverse_reference)`

Specifies one-to-one, one-to-many or many-to-many relationship between classes (tables). Both `reference` or `inverse_reference` fields should have reference or array of reference types. `inverse_reference` is a field of a referenced table containing inverse reference(s) to the current table. Inverse references are automatically updated by FastDB and also are used for query optimization (see Inverse references).

`METHOD(name)`

Specifies method of the class. Method should be instance member function, without any parameters and returning boolean, numeric, reference or string type. Methods should be specified after all other attributes of the class.

Although only atomic fields can be indexed, index type can be also specified for structures. Index will be created for component of the structure only if such type of index is specified in the index type mask of the structure. It makes possible to programmers to enable or disable indices for structure fields depending on the role of the structure in the record.

The following example illustrates creation of type descriptor:

```
class dbDateTime {
    int4 stamp;
  public:

    int year() {
        return localtime((time_t*)&stamp)->tm_year + 1900;
    }
    ...

    CLASS_DESCRIPTOR(dbDateTime,
                 (KEY(stamp,INDEXED|HASHED),
                  METHOD(year), METHOD(month), METHOD(day),
                  METHOD(dayOfYear), METHOD(dayOfWeek),
                  METHOD(hour), METHOD(minute), METHOD(second)));
};

class Detail {
  public:
    char const* name;
    char const* material;
    char const* color;
    real4       weight;

    dbArray< dbReference<Contract> > contracts;

    TYPE_DESCRIPTOR((KEY(name, INDEXED|HASHED),
                     KEY(material, HASHED),
                     KEY(color, HASHED),
                     KEY(weight, INDEXED),
                     RELATION(contracts, detail)));
};

class Contract {
  public:
    dbDateTime            delivery;
    int4                  quantity;
    int8                  price;
    dbReference<Detail>   detail;
    dbReference<Supplier> supplier;

    TYPE_DESCRIPTOR((KEY(delivery, HASHED|INDEXED),
                     KEY(quantity, INDEXED),
                     KEY(price, INDEXED),
                     RELATION(detail, contracts),
                     RELATION(supplier, contracts)));
};
```

Type descriptors should be defined for all classes used in database. In addition to defining type descriptors, it is necessary to establish mapping between C++ classes and database tables. Macro `REGISTER(name)` will do it. Unlike `TYPE_DESCRIPTOR`, `REGISTER` macro should be used in implementation file and not in header file. It constructs descriptor of the table associated with the class. If you are going to work with multiple databases from one application, it is possible to register table in concrete database by means of `REGISTER_IN(name,database)` macro. Parameter database of this macro should be pointer to `dbDatabase` object. Below is example of registration tables in database:

```
REGISTER(Detail);
REGISTER(Supplier);
REGISTER(Contract);
```

Table (and correspondent class) can be used only with one database at each moment of time. When you open database, FastDB imports all classes defined in application in database. If class with the same name already exists in database, its descriptor stored in the database is compared with descriptor of this class in application. If there are differences in class definitions, FastDB tries to convert records from the table to new format. Any kind of conversions between numeric types (integer to real, real to integer, with extension or truncation, are allowed). Also addition of new fields can be easily handled. But removing of the fields is only possible for empty tables (to avoid accidental data destruction).

After loading all class descriptors, FastDB checks if all indices specified in the application class descriptor are already present in database, constructing new indices and removing indices, which are no more used. Reformatting of table and adding/removing indices is only possible when there is no more than one application accessing database. So when first application is attached to database, it can perform table conversion. All other application can only add new classes to database, but not change existed ones.

There is one special preexisted table in database — `Metatable`, which contains information about other tables in database. C++ programmer need not to access this table, because format of database tables is specified by C++ classes. But in interactive SQL program it is possible to examine this table to get information about record fields.

3.2 Query

Class query is used for two purposes:

1. construct query and bind query parameters
2. cache compiled queries

FastDB provides overloaded `=` and `,` C++ operators to construct query statement with parameters. Parameters can be specified directly in places where they are used, eliminating any mapping between parameters placeholders and C variables. In the following example of query pointers to the parameters price and quantity are stored in the query, so that query can be executed several times with different values of parameters. C++ overloaded functions make it possible to automatically determine type of parameter, requiring no extra information to be supplied by programmer (so programmer has no possibility to make a bug).

```
dbQuery q;
int price, quantity;
q = "price >=",price,"or quantity >=",quantity;
```

As far as `char*` type can be used either for specifying part of query (such as `"price >="`) either for parameter of string type, FastDB uses special rule to resolve this ambiguity. This rule is based on the assumption that there is no reason for splitting query text in two strings like (`"price ", ">="`) or specifying more than one parameter sequentially (`"color=",color,color`). So FastDB assumes first string to be part of the query text and switches to operand mode after it. In operand mode FastDB treats char* argument as query parameter and switches back to query text mode, and so on... It is also possible not to use this "syntatic sugar" and construct query elements explicitly by `dbQuery::append(dbQueryElement::ElementType type, void const* ptr)` method. Before appending elements to the query, it is necessary to reset query by `dbQuery::reset()` method (operator = do it automatically).

It is not to possible use C++ numeric constants as query parameters, because parameters are accessed by reference. But it is possible to use string constants, because strings are passed by value. There two possible ways of specifying string parameters in query: using string buffer or pointer to pointer to string:

```
dbQuery q;
char* type;
char name[256];
q = "name=",name,"and type=",&type

scanf("%s", name);
type = "A";
cursor.select(q);
...
scanf("%s", name);
type = "B";
cursor.select(q);
...
```

Query variable can not be passed to a function as parameter or be assigned to other variable. When FastDb compiles the query, it saves compiled tree in this object. Next time the query will be used, no compilation is need and ready compiled tree can be used. It saves some time needed for query compilation.

FastDB provides two approaches of integration user-defined types in database. First — definition of class methods — was already mentioned. Another approach deals only with query construction. Programmer should define methods, which will not do actual calculations, but instead of this returns expression in terms of predefine database types, which performs necessary calculation. It is better to describe it by example. FastDB has no builtin datetime type. Instead of this normal C++ class dbDateTime can be used by programmer. This class defines methods allowing to compare two dates using normal relational operators and specify datetime field in order list:

```
class dbDateTime {
    int4 stamp;
  public:
    ...
    dbQueryExpression operator == (char const* field) {
        dbQueryExpression expr;
        expr = dbComponent(field,"stamp"),"=",stamp;
        return expr;
    }
    dbQueryExpression operator != (char const* field) {
        dbQueryExpression expr;
        expr = dbComponent(field,"stamp"),"",stamp;
        return expr;
    }
    dbQueryExpression operator < (char const* field) {
        dbQueryExpression expr;
        expr = dbComponent(field,"stamp"),"<",stamp;
        return expr;
    }
    dbQueryExpression operator <= (char const* field) {
        dbQueryExpression expr;
        expr = dbComponent(field,"stamp"),"<=",stamp;
        return expr;
    }
```

```
    dbQueryExpression operator > (char const* field) {
        dbQueryExpression expr;
        expr = dbComponent(field,"stamp"),">",stamp;
        return expr;
    }
    dbQueryExpression operator >= (char const* field) {
        dbQueryExpression expr;
        expr = dbComponent(field,"stamp"),">=",stamp;
        return expr;
    }
    friend dbQueryExpression between(char const* field, dbDateTime& from,
                                     dbDateTime& till)
    {
        dbQueryExpression expr;
        expr=dbComponent(field,"stamp"),"between",from.stamp,"and",till.stamp;
        return expr;
    }

    friend dbQueryExpression ascent(char const* field) {
        dbQueryExpression expr;
        expr=dbComponent(field,"stamp");
        return expr;
    }
    friend dbQueryExpression descent(char const* field) {
        dbQueryExpression expr;
        expr=dbComponent(field,"stamp"),"desc";
        return expr;
    }
};
```

All these method receives as their parameter name of the field in the record. This name is used to contract full name of the records components. It can be done by class `dbComponent`, which constructor takes name the the structure field and name of the component of the structure and returns compound name separated by `'.'` symbol. Class `dbQueryExpression` is used to collect expression items. Expression is automatically enclosed in parentheses, eliminating conflicts with operators precedence.

So, assuming record contains field delivery of `dbDateTime` type it is possible to construct queries like this:

```
    dbDateTime from, till;
    q1 = between("delivery", from, till),"order by",ascent("delivery");
    q2 = till >= "delivery";
```

Except these methods, some class specific method can be also defined in such way, for example method overlaps for region type. The benefit of this approach is that database engine will work with predefined types and is able to apply indices and other optimizations to proceed such query. And from the other side, encapsulation of class implementation is preserved, so programmer should not rewrite all queries when class representation is changed.

Variables of following C++ types can be used as query parameters:

```
int1   bool
int2   char const*
int4   char **
int8   char const**
real4  dbReference<T>
real8  dbArray< dbReference<T> >
```

3.3 Cursor

Cursors are used to access records returned by `select` statement. FastDB provides typed cursors, *i. e.* cursors associated with concrete tables. There are two kinds of cursors in FastDB: readonly cursors and cursors for update. Cursors in FastDB are represented by C++ template class `dbCursor<T>`, where `T` is name of C++ classes associated with database table. Cursor type should be specified in constructor of the cursor. By default read-only cursor is created. To create cursor for update, you should pass parameter dbCursorForUpdate to the constructor.

Query is executed by cursor `select(dbQuery& q)` or `select()` methods. Last method can be used to iterate through all records in the table. Both methods return number of selected records and set current position to the first record (if available). Cursors can be scrolled in forward or backward directions. Methods `next()`, `prev()`, `first()`, `last()` can be used to change current position of the cursor. If operation can not be performed (no more records available), these methods return NULL and cursor position is not changed.

Cursor for class `T` contains instance of class `T`, used for fetching current record. That is why table classes should have default constructor (constructor without parameters), which has no side effects. FastDB optimizes fetching records from database, copying only data from fixed part of the object. String bodies are not copied, instead of this correspondent field points directly in database. The same is true for arrays, which components has the same representation in database as in application (arrays of scalar types or arrays of nested structures of scalar components).

Application should not change elements of strings and arrays in database directly. When array method need to update array body, it create in-memory copy of the array and updates this copy. If programmer wants to update string field, it should assign to the pointer new value, but don't change string directly in database. It is recommended to use `char const*` type instead of `char*` for string components, to make it possible to compiler to detect illegal usage of strings.

Cursor class provides `get()` method for obtaining pointer to the current record (stored inside cursor). Also overloaded operator-¿ can be used to access components of current record. If cursor is opened for update, current record can be changed and stored in database by `update()` method or can be removed. If current record is removed, next record becomes current. If there is no next record, then previous record becomes current (if exists). Method `removeAll()` removes all records in the table and method removeAllSelected - all records selected by the cursor.

When records are updated, database size can be increased and extension of database section in virtual memory is needed. As a result of such remapping, base address of the section can be changed and all pointers to database fields kept by application will become invalid. FastDB automatically updates current records in all opened cursors when database section is remapped. So, when database is updated, programmer should access record fields only through the cursors -¿ method and do not use pointer variables.

Memory used for the current selection can be released by `reset()` method. This method is automatically called by `select()`, `dbDatabase::commit()`, `dbDatabase::rollback()` methods and cursor destructor, so in most cases there is no need to call `reset()` method explicitly.

Cursors can be also used to access records by reference. Method `at(dbReference const& ref)` sets cursor to the record pointed by the reference. In this case selection consists exactly of one record and `next()`, `prev()` methods will always return `NULL`. As far as cursors and references in FastDB are strictly typed, all necessary checking can be done statically by compiler and no dynamic type checking is needed. The only kind of checking, which is done at runtime, is checking for null reference. Object identifier of current record in the cursor can be obtained by `currentId()` method.

It is possible to restrict number of records returned by `select` statement. Cursor has two methods `setSelectionLimit(size_t lim)` and `unsetSelectionLimit()`, which can be used to set/unset limitation on number of records returned by query. In some situations programmer wants to receive only one record or only few first records, so query execution time and size of consumed memory can be reduced by limiting size of selection. But if you specify order for selected records, query with restriction for k records will no return first k records with the smallest value of the key. Instead of this arbitrary k records will be taken and then sorted.

So all operations with database data are performed by means of cursors. The only exception is insert operation. FastDB provides overloaded insert function:

```
template<class T>
dbReference insert(T const& record);
```

This function will insert record at the end of the table and return reference of the created object. Order of insertion is strictly specified in FastDB and applications can use this assumption about records order in the table. For applications widely using references for navigation between objects it is necessary to have some root object, from which traversal by references can be made. Good candidate for such root object is first record in the table (it is also the oldest record in the table). This record can be accessed by execution `select()` method without parameter. The current record in the cursor will be the first record in the table.

FastDB C++ API defines special `null` variable of reference type. It is possible to compare `null` variable with references or assign it to the reference:

```
void update(dbReference<Contract> c) {
    if (c != null) {
        dbCursor<Contract> contract(dbCursorForUpdate);
        contract.at(c);
        contract->supplier = null;
    }
}
```

3.4 Database

Class `dbDatabase` controls interaction of application with database. It performs synchronization of concurrent accesses to the database, transaction management, memory allocation, error handling,...

Constructor of `dbDatabase` objects allows programmer to specify some database parameters:

```
dbDatabase(dbAccessType type = dbAllAccess,
           size_t dbInitSize = dbDefaultInitDatabaseSize,
           size_t dbExtensionQuantum = dbDefaultExtensionQuantum,
           size_t dbInitIndexSize = dbDefaultInitIndexSize,
           int nThreads = 1);
```

Database can be opened in readonly mode (`dbDatabase::dbReadOnly` access type) or in normal mode allowing modification of database (`dbDatabase::dbAllAccess`). When database is opened in readonly mode, no new class definitions can be added to database and also definition of existed class and indices can not be altered.

Parameter `dbInitSize` specifies initial size of database file. Database file is grown on demand and setting of initial size can only reduce number of reallocations (which can take a lot of time). In current implementation of FastDB database size is at least doubled at each extension. Default value of this parameter is 4 megabytes.

Parameter `dbExtensionQuantum` specifies quantum of extension of memory allocation bitmap. Briefly speaking, value of this parameters specifies how much memory will be allocated sequentially without attempt to reuse space of deallocated objects. Default value of this parameter is 4 Mb. See section **Memory allocation** for more details.

Parameter `dbInitIndexSize` specifies initial index size. All objects in FastDB are accessed through object index. There are two copies of object index: current and committed. Object indices are reallocated on demand and setting initial index size can only reduce (or increase) number of reallocations. Default value of this parameter is 64K object identifiers.

And the last parameter `nThreads` controls level of query parallelization. If it is greater than `1`, then FastDb can start parallel execution of some queries (including sorting of result). Specified number of parallel threads will be spawned by FastDB engine in this case. Usually there is no sense to specify the value of this parameter greater than number of online CPUs in the system. It is also possible to pass zero as value of the parameter, in this case FastDB will automatically detect number of online CPUs in the system. Number of threads can be also set by dbDatabase::setConcurrency method at any moment of time.

Class `dbDatabase` contains static field `dbParallelScanThreshold`, which specifies threshold for number of records records in the table after which query parallelization is used. Default value of this parameter is 1000.

Database can be opened by `open(char const* databaseName, char const* fileName = NULL)` method. If file name parameter is omitted, it is constructed from database name by appending `".fdb"` suffix. Database name should be arbitrary identifier consisting of any symbols except `'\'`. Methodopen returns true if database was successfully opened or false if open operation failed. In last case database `handleError` method is called with `DatabaseOpenError` error code. Database session can be terminated by close method, which implicitly commits current transaction.

In multithreaded application each thread, which wants to access database, should first be attached to it. Method `dbDatabase::attach()` allocates thread specific data and attaches thread to the database. This method is automatically called by `open()` method, so there is no reason to call `attach()` method for the thread opening database. When thread finishes work with database, it should call `dbDatabase::detach()` method. Method close automatically invokes `detach()|` method. Method `detach()` implicitly commits current transaction. Attempt to access database by detached thread causes assertion failure.

FastDB is able to perform compilation and execution of queries in parallel, providing significant increase of performance in multiprocessor systems. But concurrent updates of database are not possible (this is a price for efficient log-less transaction mechanism and zero time recovery). When application wants to modify database (open cursor for update or insert new record in the table), it first locks database in exclusive mode, prohibiting accesses to database by other applications, even for read-only queries. So to avoid blocking of database application for a long time, modification transactions should be done as short as possible. No blocking operations (like waiting input from the user) should be done within transaction.

Using only shared and exclusive locks on database level, allows FastDB to almost eliminate overhead of locking and optimize speed of execution of non-conflicting operations. But if many applications simultaneously updates different parts of database, then approach used in FastDB will be very inefficient. That is why FastDB is most suitable for single-application database access model or for multiple applications with read-dominated access pattern model.

Both cursor and query objects should be used only by one thread in multithreaded application. If there are more than one threads in your applications, use local variables for cursors and queries objects in each thread. And dbDatabase object is shared between all threads and uses thread specific data to perform query compilation and execution in parallel with minimal synchronization overhead. There are few global things, which require synchronization: symbol table, pool of tree node,... But scanning, parsing and execution of query can be done without any synchronization, providing high level of concurrency at multiprocessor systems.

Database transaction is started by first select or insert operation. If cursor for update is used, then database is locked in exclusive mode, prohibiting access to the database by other applications and threads. If read-only cursor is used, then database is locked in shared mode preventing other application and threads from modifying database, but allowing concurrent read requests execution. Transaction should be explicitly terminated either by `dbDatabase::commit()` method, which fixes all changes done by transaction in database, or by `dbDatabase::rollback()` method which undo all modifications done by transaction. Method `dbDatabase::close()` automatically commits current transaction.

If you start transaction by performing selection using read-only cursor and then use cursor for update to perform some modifications of database, database will be first locked in shared mode and then lock will be upgraded to exclusive. This can cause deadlock problem if database is simultaneously accessed by several applications. Imagine that application A starts read transaction and application B also starts read transaction. Both of them hold shared locks on the database. If both of them wants to upgrade their locks to exclusive, they will forever block each other (exclusive lock can not be granted until shared lock of other process exists). To avoid such situation try to use cursor for update at the beginning of transaction or explicitly use `dbdatabase::lock()` method. More information about implementation of transactions in FastDB can be found in section **Transactions**.

It is possible to explicitly lock database by `lock()` method. Locking is usually done automatically and there are few cases when you will want to use this method. It will lock database in exclusive mode until the end of current transaction.

Backup of database can be done by `dbDatabase::backup(char const* file)` method. Backup locks database in shared mode and flush image of database in main memory to specified file. Because of using of shadow object index, database file is always in consistent state, so recovery from the backup can be performed just by renaming backup file (if backup was performed on tape, it should be first restored to the disk).

Class `dbDatabase` is also responsible for handling various application errors, such as syntax errors in query compilation, out of range index or null reference access during query execution. There is virtual method `dbDatabase::handleError`, which handles these errors:

```
virtual void handleError(dbErrorClass error,
                         char const*  msg = NULL,
                         int          arg = 0);
```

Programmer can derive his own subclass from dbDatabase class and redefine default reaction on errors.
dbDatabase class and redefine default reaction on errors.

Class	**Description**	**Argument**	**Default reaction**
`QueryError`	query compilation error	position in query string	abort compilation
`ArithmeticError`	arithmetic error during division or power operations	—	terminate application
`IndexOutOfRangeError`	index is out if array bounds	value of index	terminate application
`DatabaseOpenError`	error while database opening	—	open method will return `false`
`FileError`	failure of file IO operation	error code	terminate application
`OutOfMemoryError`	not enough memory for object allocation	requested allocation size	terminate application
`Deadlock`	upgrading lock cause deadlock	—	terminate application
`NullReferenceError`	null reference is accessed during query execution	—	terminate application

Table 4: Error classes and default handling

4 Query optimization

Execution of query when all data is present in memory is very fast, comparing with time of query execution in traditional RDBMSes. But FastDB even more increase speed of query execution by applying several optimizations: using indices, inverse references and query parallelization. The next sections supply more information about these optimizations.

4.1 Using indices in queries

Indices is traditional approach for increasing RDBMS performance. FastDB uses two types of indices: extensible hash table and T-tree. First provides the fastest way (constant time in average) to access record with specified value of the key. While T-tree, which is combination of AVL-Tree and array, has the same role for MMRDBMS as B-Tree for traditional RDBMSes. It provides search, insertion and deleting operations with guaranteed logarithmic complexity (it means that time of performing search/insert/delete operation for table with N records is $C * log_2(N)$, where C is some constant. T-tree is more suitable for MMDBMSes than B-Tree, because the last one tries to minimize number of page loads (because it is expensive operation in disk-based databases), while T-tree tries to optimize number of compare/move operations. T-tree is best to use with range operations or when records order is significant.

FastDB uses simple rules for applying indices, allowing programmer to predict when and which index will be used. Check for index applicability is done during each query execution, so decision can be made depending on values of operands. The following rules describes algorithm of applying indices by FastDB:

- Compiled condition expression is always inspected from left to right.
- If topmost expression is `AND` then try to apply index to left part of expression, using right part as filter.
- While topmost expression is OR and index can be applied to left part of the expression, then apply index and test right part for possibility to use indices.
- Otherwise, index is applicable to the expression, when
  1. topmost expression is a relational operation (`=`. `<` `>` `<=`, `>=` between `like`)
  2. type of operands is boolean, numeric or string
  3. right operand(s) of the expression is either constant literal or C++ variable, and
  4. left part is indexed field of the record;
  5. index is compatible with relational operation

Now we should make clear what phrase "index is compatible with operation" means and which type of index is used in each case. Hash table can be used when:

- Comparison for equality `=` is used.
- `between` operation is used and values of both bounds operands are the same.

  `like` operation is used and pattern string contains no special characters (`'%'` or `'_'`) and no escape characters (specified in `escape` part).

T-tree index can be applied if hash table is not applicable (or field is not hashed) and:

- Comparison operation `=`, `<`, `>`. `<=` or `>=` `between` is used.
- like operation is used and pattern string contains non empty prefix (first character of pattern is not `'%'` or `'_'`).

If index is used to search prefix of `like` expression, and suffix is not just the `'%'` character, then index search operation can return more records than really match the pattern. In this case we should filter index search output by applying pattern match operation.

When search condition is disjunction of several subexpressions (expression contains several alternatives combined by or operator), then several indices can be used for query execution. To avoid record duplicates in this case, bitmap is used in cursor to mark records already included in the selection.

If search condition requires sequential table scan, T-tree index still can be used if order by clause contains the single record field for which T-tree index is defined. As far as sorting is very expensive operation, using of index instead of sorting significantly reduce time of query execution.

It is possible to check which indices are used for query execution and number of probes done during index search be compiling FastDB with option `-DDEBUG=DEBUG_TRACE`. In this case FastDB will dump trace information about database functionality including information about indices.

4.2 Inverse references

Inverse references provides efficient and reliable way of establishing relations between tables. FastDB uses information about inverse reference when record is inserted/updated/deleted and also for query optimization. Relations between records can be of one of the following types: *one-to-one*, *one-to-many* and *many-to-many*.

- *One-to-one* relation is represented by reference field in self and target records.
- *One-to-many* relation is represented by array of references field in self record and reference field in the record of referenced table.
- *Many-to-one* relation is represented by reference field in self record and array of references field in the record of target table.
- *Many-to-many* relation is represented by array of references fields in self and target records.

When record with declared relations is inserted in the table, inverse references in all tables been in relation with this records are updated to point to this record. When record is updated and field specifying records relationships are changed, then inverse references are also reconstructed automatically, by removing references to updated record from that records which are no more in relation with this record and setting inverse references to updated record for new records included in relation. And when record is deleted from the table, references to it are removed from all inverse reference fields.

Due to efficiency reasons, FastDB is not able to guaranty consistency of all references. If you remove record from the table, there are still can be references to removed record in database. Accessing these references can cause unpredictable behavior of application and even database corruption. Using inverse references allows to eliminate this problem, because all references will be updated automatically and consistency of references is preserved.

Lets use the following table definitions as example:

```
class Contract;

class Detail {
  public:
    char const* name;
    char const* material;
    char const* color;
    real4       weight;

    dbArray< dbReference<Contract> > contracts;

    TYPE_DESCRIPTOR((KEY(name, INDEXED|HASHED),
                     KEY(material, HASHED),
                     KEY(color, HASHED),
                     KEY(weight, INDEXED),
                     RELATION(contracts, detail)));
};

class Supplier {
  public:
    char const* company;
    char const* location;
    bool        foreign;

    dbArray< dbReference<Contract> > contracts;
```

```
    TYPE_DESCRIPTOR((KEY(company, INDEXED|HASHED),
                     KEY(location, HASHED),
                     FIELD(foreign),
                     RELATION(contracts, supplier)));
};


class Contract {
  public:
    dbDateTime            delivery;
    int4                  quantity;
    int8                  price;
    dbReference<Detail>   detail;
    dbReference<Supplier> supplier;

    TYPE_DESCRIPTOR((KEY(delivery, HASHED|INDEXED),
                     KEY(quantity, INDEXED),
                     KEY(price, INDEXED),
                     RELATION(detail, contracts),
                     RELATION(supplier, contracts)));
};
```

In this example there are one-to-many relations between tables `Detail-Contract` and `Supplier-Contract`. When `Contract` record is inserted in database, it is necessary only to set references detail and supplier to correspondent records of `Detail` and `Supplier` tables. Inverse references contracts in these records will be updated automatically. The same is happened when `Contract` record is removed, references to removed record will be automatically excluded from contracts field of referenced `Detail` and `Supplier` records.

Moreover using inverse reference allows to chose more effective plan of query execution. Consider the following query selecting all details shipped by some company:

```
    q = "exists i:(contracts[i].supplier.company=",company,")";
```

The straightforward approach to execution of this query is scanning Detail table and testing each record for this condition. But using inverse reference we can choose another approach: perform index search in `Supplier` table for records with specified company name and then use inverse references to locate records from Detail table been in transitive relation with selected supplier records. Certainly we should eliminate duplicates of records, which can appear because company can ship a number of different details. This is done by bitmap in cursor object. As far as index search is significantly faster than sequential search and accessing record by reference is very fast operation, total time of such query execution is much shorter comparing with straightforward approach.

4.3 Realtime issues

FastDB is not true realtime system, because it is based on non-realtime operations systems (like NT or Unix), which mostly doesn't fit realtime requirements. But with some restrictions it is possible to give quite precise estimations for query execution time in FastDB. Such restrictions include:

1. All database is fit in physical memory and no swapping can take place during application work;
2. Other applications should not consume much CPU time and system resources (especially memory) to avoid influence on main application.
3. Required time of reaction is significantly longer than quantum of processes rescheduling in operating system.

Algorithms used in FastDB allows to quite precisely calculate average and maximal time of query execution depending on number of records in the table (in assumption that size of array fields in records are significantly smaller than table size and time of iteration through array elements can be excluded from estimation). The following table provides you with complexity of search in table with N records depending on search condition:

Type of search	**Average**	**Maximal**
Sequential search	$O(N)$	$O(N)$
Sequential search with sorting	$O(N * log(N))$	O(N*log(N))
Search using hash table	$O(1)$	$O(N)$
Search using T-tree	$O(log(N))$	$O(log(N))$
Access by reference	$O(1)$	$O(1)$

FastDB uses Heapsort algorithm for sorting selected records to provide guaranteed $log(N)$ complexity (quicksort is in average a little bit faster, but worst time is $O(N * N)$). Hash table also has different average and maximal complexity. In average hash table search is faster than search using T-tree, but in worst case it is equivalent to sequential search while T-tree always guarantee $log(N)$ complexity.

Execution of update statements in FastDB is also fast, but this time is less predictable, because commit requires flushing of modified pages to disk which can cause unpredictable operating system delays.

4.4 Parallel query execution

FastDB is able to split query into several parallel jobs, which will be executed without any contention with each other. Parallelization of query is done by FastDB only for sequential scans if the table. In this case splitting job between N processors can reduce query execution time about N times (even more if result should be sorted).

To split table scan, FastDB starts N threads; each of them tests $N - s$ record of the table (i.e. thread number 0 test records 0, N,$2*N$, ... thread number 1 test records 1, $1+N$, $1+2*N$, ... and so on). Each thread builds its own list of selected records. After termination of all threads, these lists are concatenated to construct the single result list.

If result should be sorted, then each thread, after finishing the table scan, sorts the records it selected. After termination of all threads, their lists are merged (as it is done with external sort).

Parallel query execution is controlled by two parameters: number of spawned threads and parallel search threshold. First is specified in `dbDatabase` class constructor or set by `dbDatabase::setConcurrency` method. Zero value of this parameter asks FastDB to automatically detect number of online CPUs in the system and spawn exactly this number of threads. By default number of threads is set to `1`, so no parallel query execution takes place.

Parallel search threshold parameter specifies minimal number of records in the table for which parallelization of query can improve query performance (starting of threads has its own overhead). This parameter is static component of `dbDatabase` class and can be changed by application at any moment of time.

Parallel query execution is not possible when:

1. Indices are used for query execution.
2. Number of records in the table is less than `dbDatabase::dbParallelScanThreshold`;
3. Selection limit is set for the cursor;
4. The query includes `start from` part;

5 FastDB implementation issues

This section describes some aspects of FastDB implementation. It is not necessary to read this section unless you want to know about FastDB internals.

5.1 Memory allocation

Memory allocation is performed in FASTDB by bitmap. Memory is allocated in chunks called allocation quantum. In current version of FastDB the size of allocation quantum is 16 bytes. It means that size of all allocated objects is aligned on 16 byte boundary. Each 16 byte of database memory is represented by one bit in the bitmap. To locate hole of requested size in bitmap, FastDB sequentially searches bitmap pages for correspondent number of successive cleared bits. FastDB use three arrays indexed by bitmap byte, which makes possible fast calculation of hole offset and size within the byte.

FastDB performs cyclic scanning of bitmap pages. It keeps identifier of current bitmap page and current position within the page. Each time when allocation request arrives, scanning of the bitmap starts from the current position. When last allocated bitmap page is scanned, scanning continues from the beginning (from the first bitmap page) and until current position. When no free space is found after full cycle through all bitmap pages, new bulk of memory is allocated. Size of extension is maximum of size of allocated object and extension quantum. Extension quantum is parameter of database, specified in constructor. Bitmap is extended to be able to map additional space. If virtual space is exhausted and no more bitmap pages can be allocated, then `OutOfMemory` error is reported.

Allocation memory using bitmap provides high locality of references (objects are mostly allocated sequentially) and also minimizes number of modified pages. Minimization of number of modified pages is significant when commit operation is performed and all dirty pages should be flushed on the disk. When all cloned objects are placed sequentially, number of modified pages is minimal and so transaction commit time is also reduced. Using extension quantum also helps to preserve sequential allocation. Once bitmap is extended, objects will be allocated sequentially until extension quantum will be completely used. Only after reaching the end of the bitmap, scanning restarts from the beginning searching for holes in previously allocated memory.

To reduce number of bitmap pages scans, FastDB associates descriptor with each page, which is used to remember maximal size of the hole on the page. Calculation of maximal hole size is performed in the following way: if an object of size M cannot be allocated from this bitmap pages, then maximal hole size is less than M, so M is stored in the page descriptor if previous value of descriptor is larger than M. For next allocation of object of size greater or equal than M, we will skip this bitmap page. Page descriptor is reset when some object is deallocated within this bitmap page.

Some database objects (like hash table pages) should be aligned on page boundary to provide more efficient access. FastDB memory allocator checks requested size and if it is aligned on page boundary, then address of allocated memory segment is also aligned on page boundary. Search of free hole will be done faster in this case, because FastDB increases step of current position increment according to the value of alignment.

To be able to deallocate memory used by object, FastDB needs to keep somewhere information about object size. There are two ways of getting object size in FastDB. All table records are prepended by record header, which contains record size and pointer of L2-list linking all records in the table. So size of the table record object can be extracted from record header. Internal database objects (bitmap pages, T-tree and hash table nodes) has known size and are allocated without any header. Instead of this handles of such objects contains special markers, which allows to determine class of the object and get it size from the table of builtin object sizes. It is possible to use markers because allocation is always done in quantum of 16 bytes, so low 4 bits of object handle are not used.

It is possible to create database larger than 4Gb or containing more than 4Gb of objects if you pass `dbDatabaseOffsetBits` or `dbDatabaseOidBits` parameters with values greater than 32 in compiler command line. In this case FastDB will use 8 byte integer type for representing object handle/object identifier. It will work only on truly 64-bit operating systems, like Digital Unix.

5.2 Transactions

Each record (object) in FastDB has unique identifier (OID). Object identifiers are used to implement references between objects. To locate object by reference, its OID is used as index in array of object offsets within the file. This array is called object index and element of this array — object handle. These are two copies of object indices in FastDB, one of which is current and the other a shadow. Header of database contains pointers to both object indices and indicator which index is current at this moment.

When object is modified first time, it is cloned (copy of the object is created) and object handle in current index is changed to point to newly created object copy. And shadow index still contains handle which points to the original version of the object. All changes are done with the object copy, leaving original object unchanged. FastDB marks in special bitmap page of the object index, which contains modified object handle.

When transaction is committed, FastDB first checks if size of object index was increased during current transaction. If so, it also reallocates shadow copy of object index. Then FastDB frees memory for all "old objects", i.e. objects which was cloned within transaction. Memory can not be deallocated before commit, because we wants to preserve consistent state of the database by keeping cloned object unchanged. If we deallocate memory immediately after cloning, new object can be allocated at the place of cloned object and we loose consistency. As far as memory deallocation is done in FastDB by bitmap using the same transaction mechanism as for normal database objects, deallocation of object space will require clearing some bits in bitmap page, which also should be cloned before modification. Cloning bitmap page will require new space for allocation the page copy, and we can reuse space of deallocated objects. And it is not acceptable due to the reason explained above — we will loose database consistency. That is why deallocation of object is done in two steps. When object is cloned, all bitmap pages used for marking objects space, are also cloned (if not not cloned before). So when transaction is committed, we only clear bits in bitmap pages and no more requests for allocation memory can be generated at this moment.

After deallocation of old copies, FastDB flushes all modified pages on disk to synchronize content of the memory and disk file. After that FastDB changes current object index indicator in database header to switch roles of the object indices. Now object index, which was current becomes shadow, and shadow index becomes current. Then FastDB again flushes modified page (i.e. page with database header) on disk, transferring database to new consistent state. After that FastDB copies all modified handles from new object index to object index which was previously shadow and now becomes current. At this moment contents of both indices is synchronized and FastDb is ready to start new transaction.

Bitmap of modified object index pages is used to minimize time of committing transaction. Not the whole object index, but only its modified pages should be copied. After committing of transaction bitmap is cleared.

When transaction is explicitly aborted by `dbDatabase::rollback` method, shadow object index is copied back to the current index, eliminating all changes done by aborted transaction. After the end of copying, both indices are identical again and database state corresponds to the moment before the start of current transaction.

Allocation of object handles is done by free handles list. Header of the list is also shadowed and two instances of list headers are stored in database header. Switch between them is done in the same way as switch of object indices. When there are no more free elements in the list, FastDB allocates handles from the unused part of new index. When there is no more space in the index, it is reallocated. Object index is the only entity in database which is not cloned on modification. Instead of this two copies of object index are always used.

There are some predefined OID values in FastDB. OID `0` is reserved as invalid object identifier. OID `1` is used as identifier of metatable object — table containing descriptors of all other tables in database. This table is automatically constructed while database initialization and descriptors of all registered application classes are stored in this metatable. OID starting from `2` are reserved for bitmap pages. Number of bitmap pages depends on database maximum virtual space. For 32 bit handles, maximal database virtual space is 4Gb, and number of bitmap pages can be calculated as this size divided by page size divided by allocation quantum size divided by number of bits in the byte. For 4 Gb virtual space, 4 Kb page size and 16 byte allocation quantum, 8K bitmap pages are required. So 8K handles are reserved in object index for bitmap. Bitmap pages are allocated on demand, when database size is extended. So OID of first users object will be 8194.

5.3 Recovery

Recovery procedure is trivial in FastDB. There are two instances of object index, one of which is current and another corresponds to consistent database state. When database is opened, FastDB checks database header to detect if database was normally closed. If not (dirty flag is set in database header), then FastDB performs database recovery. Recovery is very similar to rollback of transaction. Indicator of current index in database object header is used to determine index corresponding to consistent database state and object handles from this index are copied to another object index, eliminating all changes done by uncommitted transaction. As far as the only action performed by recovery procedure is copying of objects index (really only handles having different values in current and shadow indices are copied to reduce number of modified pages) and size of object index is small, recovery can be done very fast. Fast recovery procedure reduces "out-of-service" time of application.

There is one hack which used in FastDB to increase database performance. All records in the table are linked in L2-list, allowing efficient traversal through the list and insertion/removing of records. Header of the list is stored in table object (which is record of Metatable table). L2-list pointers are stored at the beginning of the object together with object size. New records are always appended in FastDB to the end of the list. To provide consistent inclusion in database list we should clone last record in the table and table object itself. But record size can be big enough, so cloning of last record for each inserted record can cause significant space and time overhead.

To eliminate this overhead FastDB do not clone last record allowing temporary inconsistency of the list. In which state will be list if system fault happens before commit of the transaction ? Consistent version of table object will point to the record which was last record in previous consistent state of database. But as far as this record was not cloned, it can contain pointer to next record, which doesn't exist in this consistent database state. To fix this inconsistency, FastDB checks all tables in database during recovery procedure and if last record in the table contains not null next reference, it is changed to null to restore consistency.

If database file was corrupted on disk, the only way of database recovery is to use backup file (certainly if you do not forget to make it). Backup file can be made by interactive SQL utility using backup command or from application by dbDatabase::backup() method. It creates snapshot of database in specified file (it can be name of a device, tape for example). As far as database file is always in consistent state, the only think needed to perform recovery from the backup file is to replace original database file with backup file. If backup was stored at tape or some other external device, it should be first extract to the disk.

If some of application starts transaction, locks database and then crashes, then database is left in locked state and no other application can access it. To restore from this situation you should stop all applications working with database. First application opening the database after this will initialize database monitor and perform recovery after crash.

5.4 Hash table

Hash tables provides the fastest in average way of location record in the table by key. Depending on hash table size and quality of hash function one or more probes will be needed to locate the record. Hash tables are the best when most of the queries use equality comparison for record fields.

FastDB uses extensible hash table with collision chains. Hash table is implemented as an array of object references, contains pointers to collision chain. Collision chain elements form L1-list, and each element contains pointer to next element, value of hash function and OID of associated record. Hash tables can be created for boolean, numeric and string fields.

Size of hash table is automatically increased when table becomes full to prevent growth of collision chains. In current implementation hash table is extended when both of the following two conditions are true:

1. Number of records in the tables becomes greater than the hash table size.

2. Number of used elements in the hash table (number of non-empty collision chains) is greater than 2/3 of the hash table size.

First condition allows to located each element in hash table using one probe in average (certainly it depends on distribution of key values and hash function). And second condition prevents hash table from growing when capacity of possible key values set becomes smaller then hash table size (for example, there is no sense to extend hash table used for hashing char field, because no more than 256 items of hash table can be used. Each time hash table is extended, its size is doubled. More precisely hash table size is 2^{n-1}. Using odd number for hash size allows to improve quality of hashing (best is to use prime numbers for hash size) and efficiently allocates space for hash table (size of hash table is aligned on page boundary). If hash table size is 2^n, than we will always loose least n bits of hash key.

FastDB uses very simple hash function, which despite to its simplicity can provide goods results (uniformal distribution of values within hash table). Hash code is calculated using all bytes of key value by the following formula:

```
h = h*31 + *key++;
```

Hash table index is the reminder of division hash code by hash table size.

5.5 T-tree

In the article "A study of index structures for main memory database management systems" T. J. Lehman and M. J Carey proposed T-trees as a storage efficient data structure for main memory databases. T-trees are based on AVL trees proposed by Adelson-Velsky and Landis. In this subsection, we provide an overview of T-trees as implemented in FastDB.

Like AVL trees, the height of left and right subtrees of a T-tree may differ by at most one. Unlike AVL trees, each node in a T-tree stores multiple key values in a sorted order, rather than a single key value. The left-most and the right-most key value in a node define the range of key values contained in the node. Thus, the left subtree of a node contains only key values less than the left-most key value, while the right subtree contains key values greater than the right-most key value in the node. A key value which is falls between the smallest and largest key values in a node is said to be bounded by that node. Note that keys equal to the smallest or largest key in the node may or may not be considered to be bounded based on whether the index is unique and based on the search condition (e. g. "greater-than" versus "greater-than or equal-to").

A node with both a left and a right child is referred to as an internal node, a node with only one child is referred to as a semi-leaf, and a node with no children is referred to as a leaf. In order to keep occupancy high, every internal node has a minimum number of key values that it must contain (typically $k-2$, if k is the maximum number of keys that can be stored in a node). However, there is no occupancy condition on the leaves or semi-leaves.

Searching for a key value in a T-tree is relatively straightforward. For every node, a check is made to see if the key value is bounded by the left-most and the right-most key value in the node; if this is the case, then the key value is returned if it is contained in the node (else, the key value is not contained in the tree). Otherwise, if the key value is less than the left-most key value, then the left child node is searched; else the right child node is searched. The process is repeated until either the key is found or the node to be searched is null.

Insertions and deletions into the T-tree are a bit more complicated. For insertions, first a variant of the search described above is used to find the node that bounds the key value to be inserted. If such a node exists, then if there is room in the node, the key value is inserted into the node. If there is no room in the node, then the key value is inserted into the node and the left-most key value in the node is inserted into the left subtree of the node (if the left subtree is empty, then a new node is allocated and the left-most key value is inserted into it). If no bounding node is found then let N be the last node encountered by the failed search and proceed as follows: If N has room, the key value is inserted into N; else, it is inserted into a new node that is either the right or left child of N, depending on the key value and the left-most and right-most key values in N.

Deletion of a key value begins by determining the node containing the key value, and the key value is deleted from the node. If deleting the key value results in an empty leaf node, then the node is deleted. If the deletion results in an internal node or semi-leaf containing fewer than the minimum number of key values, then the deficit is made up by moving the largest key in the left subtree into the node, or by merging the node with its right child.

In both insert and delete, allocation/deallocation of a node may cause the tree to become unbalanced and rotations (RR, RL, LL, LR) may need to be performed. (The heights of subtrees in the following description include the effects of the insert or delete.) In the case of an insert, nodes along the path from the newly allocated node to the root are examined until either

1. a node for which the two subtrees have equal heights is found (in this case no rotation needs to be performed), or

2. a node for which the difference in heights between the left and the right subtrees is more than one is found and a single rotation involving the node is performed.

In the case of delete, nodes along the path from the de-allocated node's parent to the root are examined until a node is found whose subtrees' heights now differ by one. Furthermore, every time a node whose subtrees' heights differ by more than one is encountered, a rotation is performed. Note that de-allocation of a node may result in multiple rotations.

6 Interactive SQL

Interactive SUBSQL utility allows to browse any FastDB database and to perform some administration functions with it. Also import/export of data from database can be done by SUBSQL utility. Name SUBSQL was chosen to express that only subset of SQL is implemented by this utility. It is not possible to create new tables and update records in the table using SUBSQL utility. This restriction is mostly caused by complex format of FastDB table (nested records, arrays, ...) and positioning of FastDB mostly as embedded in application database engine. So SUBSQL is best for inspecting of databases and doing some administrative work, such as performing database backups.

The following rules in BNF-like notation specifies grammar of SUBSQL directives:

directive ::=
  `select (*) from` *table-name select-condition* `;`
| `insert into` *table-name* `values` *values-list* `;`
| `create index on` *field-name* `;`
| `drop` *index field-name* `;`
| `drop` *table-name*
| `open` *database-name (database-file-name)* `;`
| `delete from` *table-name*
| `backup` *file-name*
| `commit`
| `rollback`
| `exit`
| `help`

table-name ::= *identifier*
values-list ::= *tuple , tuple*
tuple ::= `(` *value* { *, value*} `)`
value ::= *number* | *string* | `true` | `false`
  | *tuple*
index ::= `index` | `hash`
field-name ::= *identifier* { `.` *identifier*}
database-name ::= *string*
database-file-name ::= *string*
file-name ::= *string*

SUBSQL automatically commits read-only transaction after each select statement in order to release shared database lock as soon as possible. But all database modification operations should be explicitly committed by commit statement or undone by rollback statement. Directives open and exit first closes opened database (if it was opened) and so implicitly commits last transaction. If database file name was not specified in open statement, then file name is constructed from database name by appending `".fdb"` suffix.

Select statement always print all record fields. FastDB doesn't support tuples, and result of the selection is always set of objects (records). Format of select statement output is similar with one accepted by insert statement (with exception of reference fields). So it is possible to export/import database table without references by means of select/insert directives of SUBSQL.

Select statement prints references in format `"#hexadecimal-number"`. But it is not possible to use this format in `insert` statement. As far as object references are represented in FastDB by internal object identifiers, reference field can not be set in insert statement (when objects are inserted in database they will be assigned new OID, so there is not sense in specifying reference field in insert statement). To ensure database reference consistency, FastDB just ignores reference fields when new records are inserted in the table with references. You should specify value `0` at the place of reference fields. If you omit `'*'` symbol in select statement, FastDB will output object identifiers of each selected record.

It is necessary to provide values for all records fields in insert statement, no default values are not supported. Components of structures and arrays should be enclosed in parentheses.

It is not possible to create or drop indices and tables while other applications are working with database. Such operations change database scheme and after such modification other applications state will become incorrect. But delete operation doesn't change database scheme, so it can be performed as normal transaction, when database is concurrently used by several applications. If SUBSQL hangs trying to execute some statement, then some other application holds the lock on database, preventing SUBSQL from accessing it.

7 Examples of FastDB applications

FastDB is shipped with some examples of database applications, which can help you create your own applications.

7.1 Example: game "Guess an animal"

"Guess an animal" is a very simple program which uses database to store the game tree. Having very simple algorithm this program shows some elements of "artificial intelligence". The more information you provide to this game, the smarter will be its behavior.

This is example of "navigation-only" application — no queries are used in this application at all. All navigation between records (object) is done by means of references. Really this application is more suitable for object oriented databases, but I include it in FastDB

1. to illustrate how easy navigation by references can be done in FastDB,
2. because this program is some kind of my "trademark" which I use in all my databases.

7.2 Example: various types of queries

This application illustrates using of various types of database queries and advantages of using inverse references. Classical Detail/Contract/Supplier database scheme is used in this example. Compare number of lines in this file with number of lines needed to implement this example using ODBC or other RDBMS C API.

7.3 Performance test

This test allows to receive some results about FastDB performance for all basic operations. This test inserts N records in the table, then performs searches using T-tree and hash table, then sequential search and sequential search with sorting. It is possible to specify level of query parallelization in command line. By default no parallelization is used. The following table contains results for some systems and $N = 1000000$. Values in the table rows specifies number of milliseconds consumed by the operation which is calculated by dividing time returned by `testperf` program by number of iterations.

System	Number of CPUs	Number of threads	Insertion[1]	Hash table search	T-tree search	Sequential search	Sequential search with sorting
Pentium-II 300, 128 Mb RAM, Windows NT	1	1	0.056	0.015	0.041	1 400	25 000
Pentium-II 350, 512 Mb RAM, Linux	1	1	0.052	0.016	0.045	1 600	33 000
Pentium-Pro 200, 128 Mb RAM, Windows NT	2	1	0.071	0.023	0.052	1 600	35 000
Pentium-Pro 200, 128 Mb RAM, Windows NT	2	2	0.071	0.023	0.052	1 800	23 000
AlphaServer 2100, 250 Mhz, 512 Mb RAM, Digital Unix	2	1	0.250	0.031	0.084	2 600	42 000
AlphaServer 2100, 250 Mhz, 512 Mb RAM, Digital Unix	2	2	0.250	0.031	0.084	1 600	23 000
AlphaStation, 500 Mhz, 256 Mb RAM, Digital Unix	2	1	0.128	0.010	0.039	1 300	36 000

It will be nice if you can run this test at some other platforms and send me results. I need to notice, that for $N = 1000000$ you need at least 128Mb of memory, otherwise you will test performance of your disks.

8 Quick start

When you are developing application for FastDB, you should first decide which data and which classes you want to store in database. Then you should describe format of database tables. Section **Table** describes how to create type and table descriptors. Do not forget to register table descriptors by `REGISTER` macro (it should be done in some implementation module). If you are going to redefine default FastDB error handler (for example, if you want to use message window for reporting instead of stderr), you should define your own database class and derive it from `dbDatabase`. You should create instance of database class and make it accessible to all application modules.

Before you can do something with database, you should open it. By checking of `dbDatabase::open()` return code you can understand if database was successfully opened. Errors during database opening doesn't cause application termination (but they are reported) even with default error handler.

Once you are certain that database is normally opened, you can start to work with database. If your application is multithreaded and several threads will work with the same database, you should attach each thread to the database by `dbDatabase::attach` method. Before thread termination, it should detached itself from database by invoking `dbDatabase::detach()` method. If your application uses navigation through database objects by references, you need some kind of root objects which can be located without any references. Best candidate for the root objects is the first record of the table. FastDB guarantee that new records are always inserted at the end of the table. So first table record is also the oldest record in the table.

To access database data you should create a number of `dbQuery` and `dbCursor` objects. If several threads are working with database, each thread should have its own instances of query and cursor objects. Usually it is enough to have one cursor for each table (or two if your application also can update table records). But in case of nested queries, using of several cursors may be needed. Query objects are usually created for each type of queries. Query objects are used also for caching compiled queries, so it will be good idea to extend live area of query variables (may be make them static).

There are four main operations with database: insert, select, update, remove. First is done without using cursors, by means of global overloaded template function insert. Selection, updating and deleting of records is performed using cursors. To be able to modify table you should use cursor for update. Cursor in FastDB is typed and contains instance of object of table class. Overloaded `->` operator of the cursor can be used to access components of current record and also to update these components. Method `update` copies data from cursor's object to the current table record. Cursor's method remove will remove current cursor record, method `removeAllSelected` will remove all selected records and method `removeAll` will remove all records in the table. Each transaction should be either committed by `dbDatabase::commit()` or aborted by `dbDatabase::rollback()` method. Transaction is started automatically when the first select, insert or remove operation is executed.

Before exiting from your application do not forget to close database. Also remember, that method `dbDatabase::close()` will automatically commit last transaction, so if it is not what you want, then explicitly perform `dbDatabase::rollback` before exit.

So template of FastDB application can look something like this:

```
//
// Header file
//
#include "fastdb.h"

extern dbDatabase db; // create database object

class MyTable {
    char const* someField;
    ...
  public:
    TYPE_DESCRIPTOR((someField));
};

//
// Implementation
//
REGISTER(MyTable);

int main()
{
    if (db.open("mydatabase")) {
        dbCursor<MyTable> cursor;
        dbQuery q;

        char value[bufSize];

        q = "someField=",value;
        gets(value);
        if (cursor.select(q) > 0) {
            do {
                printf("%s\n", cursor->someField);
            } while (cursor.next());
        }
```

```
        db.close();
        return EXIT_SUCCESS;
    } else {
        return EXIT_FAILURE;
    }
}
```

To compile FastDB application you need to include header file `"fastdb.h"`. This header file includes other FastDB header files, so make sure that FastDB directory is in compiler include directories list. To link FastDB application you need FastDB library (`"fastdb.lib"` for Windows or `"libfastdb.a"` for Unix). You can either specify full path to this library or place it in some default library directory (for example `/usr/lib` for Unix).

To build FastDB library just type make in FastDB directory. There is no autoconfiguration utility included in FastDB distribution. Most system dependent parts of code are compiled using conditional compilation. There are two makefiles in FastDB distribution. One for MS Windows with MS Visual C++ (`makefile.mvc`) and another one for generic Unix with gcc compiler (`makefile`). If you want to use Posix threads or some other compiler, you should edit this makefile. There is also make.bat, which just spawns `nmake -f makefile.mvc` command. Target `install` target in Unix makefile will copy FastDB header files, FastDB library and `subsql` utility to directories specified by `INCSPATH`, `LIBSPATH` and `BINSPATH` makefile variables correspondingly. Default values of this variables are the following:

```
INCSPATH=/usr/include
LIBSPATH=/usr/lib
BINSPATH=/usr/bin
```

Once your application starts to work, you will be busy with support and extension of you application. FastDB is able to perform automatic scheme evaluation for such cases as adding new field to the table and changing type of the field. Programmer can also add new indices or remove rarely used indices. Database trace can be switched on (by recompilation FastDB library with `-DDEBUG=DEBUG_TRACE` compiler options) to perform analysis of database functionality and efficiency of using indices.

`subsql` utility can be used for database browsing and inspection, performing online backups, importing data to and exporting data from database. FastDB will perform automatic recovery after system or application crash, you should not worry about it. The only thing you can have to do manually is stopping all database application if one of them is crashed leaving database blocked.

9 Distribution terms

I will provide e-mail support and help you with development of FastDB applications.

Look for new version at my homepage	E-Mail me about bugs and problems
`http://www.ispras.ru/~knizhnik`	`knizhnik@altavista.net`