Streaming multimedia files from relational database

Tomasz Rybak[1]

Abstract: Article presents solution for retrieving multimedia files stored in database. It starts with discussion of problems that may arise when such files are store in database. Next it presents GStreamer library that can be used in programs to access multimedia data. Then it describes system implemented with help of GStreamer to store and manage database of multimedia files. At the end article presents possible ways of enhancing created system.

Keywords: multimedia data, relational database, streaming, network

1. Introduction

Thanks to technological advances we have many devices capable of recording and playing multimedia data. Digital cameras capable of recording photos and short movies are common. Cellular phones can record and play audio and video files. Digital audio players have memory capable of storing days and days of songs; more sophisticated ones are able to cope with video files or serve as digital dictaphones.

MyLifeBits is idea proposed by Microsoft Research (Gemmel 2006) to store and manage data recorded by GPS (Global Positioning System) devices, digital cameras, cellular phones, etc. Mary Czerwinski et al. describe (Czerwinski 2006) legal, social, and technical problems and advantages and disadvantages of recording every aspect of life.

Not only personal, but also professional (for example medical) data is digitised. Data and results of experiments are stored in digital form, which makes backup copies and providing access to other professionals easier. Some hospitals even give patients records of USG scans. But such digitised health data poses many challenges, especially to patients. As noted by Wanda Pratt et al. (Pratt 2006) patients need to integrate this data with their Personal Information Management (PIM) systems and manage it together with other aspects of their lives. But, at the same time, such data need to be treated differently, as it is very personal and highly sensitive information.

As number of stored multimedia files grow, problems with managing them emerge. Different programs, like Beagle, Apple Spotlight or Google Desktop Search, try to help us with this. For now these programs only search all files stored on hard drive and are able to give results of simple queries. But soon they should also be able help us with managing stored data. Currently Beagle uses SQLite database to store information about files.

Alhough rest of article focuses on movies, similar problems can be seen when trying to access multimedia data of other types.

[1] Faculty of Computer Science, Bialystok Technical University, 15-351 Bialystok
e-mail: `rybak@ii.pb.bialystok.pl`

2. Multimedia databases

Relational model is primary model used in commercial databases and data-processing applications (Silbershatz 2004). This model assumes that each data record stored in tables has the same number of fields, and each field is atomic (first normal form, described by William Kent in Kent 1983). Also, design of most databases assumes that each row is rather small, and rows occupying more than few kilobytes are rare.

Multimedia does not hold any of those assumptions. Multimedia files are huge; a movie lasting 90 minutes can take up to 9GB in DVD standard, and up to 20GB in BluRay or HD-DVD standard. Although personal files do not take such amounts of space, each of them can take easily few dozens of megabytes.

Also, movies and other multimedia files have internal structure. According to mentioned rules of normalisation this means that they need to be divided into smaller, atomic fragments, and such fragments should be stored in database. But this is rather complicated in multimedia files. What is the most basic, atomic value in case of movie? Is it scene, cut, frame, single pixel, single value (RGB, or CrCbY)? Dividing movie into such small pieces also means that enormous work will be needed during importing files into a databases. Joining small items back into multimedia stream may be impossible in real-time because of computational constraints.

2.1. Storage methods

Usually single multimedia file is stored as single row, as it offers necessary compromise between theoretical rules and constraints given by available database systems, even though this means that it is not possible to use all operators and tools provided by database to operate on objects inside file.

Modern databases offer two ways of storing large files.

One way is to store file as additional column. Such column has special type, able to store array of bytes. Usually it is called BLOB, which stands for Binary Large OBject. The same mechanisms that are used to store and manage ordinary data types are used to manage BLOBs. This relieves programmers and database administrators from worrying about removing files when rows are removed or updated. Also, as everything is stored inside database, procedures used to make backup copy need not to be changed. But internal structures used in databases to store and manage data usually limits size of stored files to 1 or 2GB.

Second method of storing large files is to store them in file system, and to put only file name, path, or other way of identifying file into database. In this solution, only operating system limits size of stored file. But, as files are stored outside database, backup copies must take them into consideration. Special care must be given to managing files, as updating or deleting row does not removes or changes file. This creates risk of having orphaned files, i.e. files that are stored in file system, but no row points to them. Another set of problems may arise when more that one row is allowed to point to particular file. These problems may be solved by using triggers, but this is additional code that must be written, tested and maintained.

One also needs to allow programmers to access such stored files. This can be done by allowing them to access file system directly, which poses security risk. To mitigate this risk one needs to create the same access management rules and rights that are present in database to file system. This is not trivial task.

Fact that database process is running with restricted rights (usually as restricted user role) must be taken into consideration during creating file access system. To be able to respond to queries, database must be able to access multimedia files. In other words, user as which database process is running needs to be able to access directories and to read files. Usually this means that everybody can access directory in which multimedia files are stored.

Other solution is to write additional stored procedures for managing files, but this code may also be rather complicated when taking into consideration access rights.

Some databases offer solution to problem with storing files that joins both described methods, i.e. they are able to manage files stored in file system. PostgreSQL has so Large Object (LO) type. These are object identified by number OID (Object IDentifier) that can be accessed by using SQL command. They are stored not directly in rows, but as other, distinct files; row contains only OID of such file. Unfortunately, limit of 1GB size still remains, as PostgreSQL stores such files using it's own TOAST storage structures.

According to literature (Oria 2004), only metadata is stored in database. Files are located in file system, and in database pointer to file (usually name) and other data describing file is stored. But this means that code that will take multimedia file, calculate needed metadata and properties, and import import it into database, must be written. The more type of data can be imported, the more importing code need to be written.

2.2. Queries and data retrieval

Each multimedia file has properties that can be used as attributes in query. Some can be accessed directly (length or metadata like title or author that are stored as text in file) and other must be calculated. The most popular calculated metadata are histograms, average brightness, number of shots, scenes, semantic description of file. Usually (Oria 2004), as calculations may take long time, metadata is calculated during importing of file, and stored in row describing this file.

During query those parameters are used to compute distance or each file from desired result, and files which distance is less than threshold (or N closest ones) are returned. More sophisticated solution can return not entire file, but fragment of it: few frames, stream lasting given time, abstract or summation of file.

But whatever is returned, the most interesting data for client is multimedia data that can be played. The easiest to implement and fastest solution is to provide access to data by binary fragment (block) of given size. This solution however ignores features of returned data. It does not allow for transferring arbitrary number of frames or seconds of file, especially when compressed data is transmitted.

When first byte of transferred block is in the middle of the file, very probable is that first byte of block will be not first byte of frame; this means that initial fragment will have to be ignored to get to proper frame. Also time at which received fragment is located is not known.

So this solution can be used only to transfer full files. Client must receive entire file by retrieving all blocks, then join them locally into proper file, and only them can start playing. This requires large space at client, and cannot be used to play in real time.

Other, preferred solution is to provide system with time of beginning and end (or number of frames) of fragment of needed file. But when transferring compressed data it may not be possible to predict how many bytes such fragment will take, and transferring without compression is impossible due to limited capabilities of most networks. Also, as movies tend to be compressed by storing full frame once every N seconds, and other frames only as differences to next/previous one, such way may require many calculations. This also means that code able to understand all stored types of multimedia data need to be written and put into database functions.

Nonetheless, standard solution for retrieving multimedia data from databases is needed, especially if many different programs need to use stored data.

3. Multimedia data as streams

Multimedia files can be seen as continuous streams of data (Alty 2004). This approach allows for easy transferring of data, especially in the Internet, even in case of heterogeneous systems and different connection speeds between connecting parties (Rejaie 2006). It allows for easy analysis and modelling, not only in case of multimedia streams but, as used by Rodrigo Fonseca et al. (Fonseca 2005), for analysis of any streams of data.

Streaming approach to multimedia need not to be restricted to networking. It can be used in case of programs running on local machine, and even inside single application.

3.1. GStreamer multimedia framework

GStreamer is generic-purpose framework for working with multimedia streams. Ir is implemented in C language, and has bindings for many languages, like Python, Java, C#. It serves as base for multimedia usage in GNOME Desktop Environment, and is used among other multimedia systems in KDE. It is component-based solution that helps programmers to write multimedia-enabled code.

Basic GStreamer object is pipeline. Pipeline is build from blocks (plugins) and is used to transform multimedia stream. Plugins are object-oriented, so they can inherit from one another.

Each block can have one or more pads, used to transfer data between plugins. Each pad has direction (input or output, in GStreamer sink or source) and capabilities (caps). Capabilities describe what type of data can be transferred. Some pads can accept many types of multimedia data, some only one. Description of plugin, i.e. it's capabilities, features, properties, signals, and pads, as well as list of all available plugins, can be accessed by any program using GStreamer library.

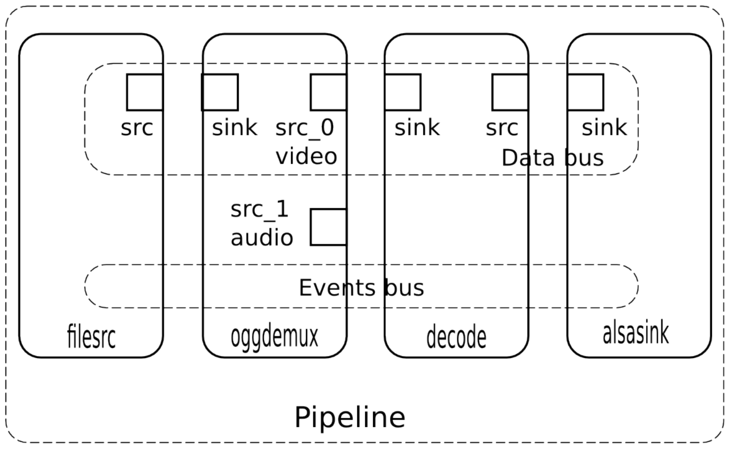

Figure 1. GStreamer sample pipeline

Plugin is called source or sink it it has only one pad; whether it is source or sink depends on direction of it's pad. Most of plugins have more than one pad, so they transform stream coming through them. Transferring means compressing and decompressing, encoding, changing resolution, frequency, etc. Some plugins have more that one source or sink pad. That are multiplexers and demultiplexers. They are used to divide movie into two separate audio and video streams, or to join two separate streams into one.

When analysing and reasoning about streams, one can seek analogies with terminology used by Fonseca et al. (Fonseca 2005). Source plugins can be seen as servers, sink plugins as clients, and other (filter) plugins as proxies.

GStreamer allows for processing many streams, which means that pipeline need not to be linear. Multiplexers and demultiplexers serve as points where different streams are joint or separated.

Each pipeline has two buses. Main bus is used to transfer multimedia data and is one-directional. Data travels from sources to sinks. Other bus is used to transfer events like end-of-stream, change of state of any of plugins, or seek event between plugins. This bus is two-directional, as any plugin can be source of event and such event must be transferred to all other objects. This bus is also used to transfer events and orders between application and GStreamer subsystem, and to synchronise plugins. Synchronisation is very important. Fragments of pipelines can be run in different threads, and all threads are synchronised it it is necessary to avoid any gaps and glitches in played content. Each pipeline can be seen as having two streams, one with data and one with events.

Sample pipeline is shown in Figure 1, with both data and event buses.

Pipeline can be created and used in two ways. Either all objects necessary to build pipeline are created in program, and joined by using code, and this way provides programmer with control over pipeline. Or one can use toll gst-launch, which takes description of pipeline, and creates and runs it. But this does not allow for controlling pipeline, as it is run by different program.

3.2. Dynamic objects

As written previously, all plugins have at least one pad. But sometimes not all pads are available all the time. For example, when demultiplexing silent movie audio stream is not present. On the other hand, some movies have many audio (for example in different languages) and video (different camera angles in case of DVD) streams.

In such cases, GStreamer allows for plugins to have dynamic pads. Each plugin that can have changing list of pads emits signal (usually "new-pad") when new pad is created. When such event occurs, user-defined function can be called, for example to build necessary fragment of pipeline.

As some parts of pipelines can be very similar, and to avoid unnecessary repetition, GStreamer allows for creating fragments of pipelines that are treated as single plugins. Such fragments, consisting of many plugins, are called bins. They also can be parts of other bins; it is implementation of composite pattern shown in book of Gang-of-Four (Gamma 1994). Each bin contains pipeline. Main pipeline is treated as a bin of the highest level. Bins can be used many times, just like plugins. As bin is treated like plugin, it must have pads. Bin pads are called ghost pads, as they not belong to bin; usually bin presents pads belonging to internal plugins as it's own.

Main advantage of using bins is that it's internal pipeline is hidden and can be changed without forcing any changes outside of bin as long as ghost pads are not changed. This allows for changing pipeline that can be adapted to different types of multimedia files. This feature is used by plugins decodebin and playbin. Decodebin can accept any type of data, and according to this type uses appropriate decoding plugins, so rest of pipeline can use raw data. Playbin can take any file or networking source and play it; internally it uses at least one decodebin plugin. It can serve as entire pipeline, as it can even use plugins for emitting data to screen or headphones.

3.3. Non-linear streams

Non-linear source can take many streams and join them together so they appear like they come from one stream. Among other plugins, GStreamer contains GNonLim, GStreamer NonLinear plugins. Main purpose of those plugins is to help with writing software for non-linear editing multimedia files, like movie editors PiTiVi and Diva, and audio editor Jokosher. But GNonLin is only source; it cannot be used in later fragments of pipeline, for filtering or as target.

All GNonLin plugins must be used inside gnlcomposition plugin, which servers as bin plugin. It manages it's internal sources, and creates necessary decodebin plugins, so from the outside it looks like gnlcomposition is source of raw data. As

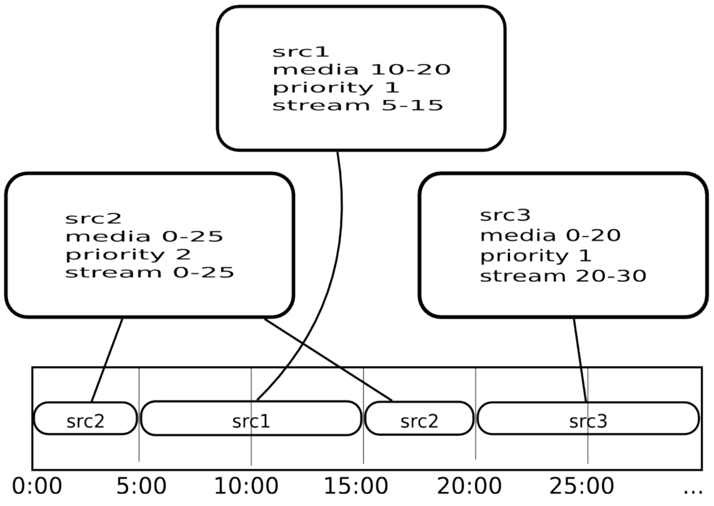

Figure 2. GNonLin media stream

there can be many different types of stream served, GNonLin can create dynamic pads and emit signal "pad-added".

As GNonLin is used to joins different streams, there are many source plugins inside gnlcomposition. Plugins that inherit from gnlsource plugin may serve as sources for GNonLin. Gnlsource is generic plugin, that serves as common base for all possible non-linear sources. Currently there exists one specific source plugin, gnlfilesource. It is possible to write own plugins, taking multimedia data from network or sound card, by using gnlsource plugin as base.

Each source plugin has many properties. These properties describe which fragment of file will be read and put into which fragment of stream. This means that many fragments can be interleaved. Also, sources have priorities which point which one is used in case of conflict between them, if more that one can be used because as source in given moment. Also sources can serve media at different than original speed, so it can be played slower or faster than original. It can be used to provide effects of transition between different sources.

Example resulting stream, taken from three sources, is shown in Figure 2.

4. Implementation

Implemented system consists of two parts. One is PostgreSQL database with stored procedure responsible for returning multimedia file being result of query. Procedure is written in Python and uses GStreamer. Second part of system is client, also written in Python. It connects to database, asks user for name of multimedia stream, sends query to database and plays returned file.

4.1. Description of used software

As written in previous section, GStreamer can be used to access multimedia data, so it is not necessary to implement own multimedia framework.

Python is open source object-oriented interpreted language. As is easy to use scripting language that doesn't require compilation, new ideas can be implemented very fast. Python allows for using many threads, but interpreter running code is single-threaded and has one lock, so all threads are interleaved during execution. No two threads in current Python implementation can be run concurrently. Both GStreamer and PostgreSQL libraries can be used in Python.

PostgreSQL is object-relational database. Although it offers two ways for storing large files, bytea (array of bytes, known as BLOB) and lo (Large Object), current version of system does not uses them because of limitations described in Section 2.1.

PostreSQL allows for creating own types. Stored procedures and functions can be used as triggers, or as operators extending existing and new types. Functions serving as aggregate functions or to cast values from one type to another can also be used.

Functions and stored procedures can be written in languages other that SQL. There exist PL/PgSQL, PL/Python, PL/Perl, PL/Tcl, PL/sh, PL/R (statistical analysis language).

Non-SQL languages usually come in two flavours. One is normal version of language, and it's name usually ends with letter "u", for example "plpythonu". It means that language is insecure, because it allows using any functionality that exist in it, together with potentially dangerous, like accessing files, processes, network, etc. Only superusers can create functions in this flavour, and run them. Option "WITH SECURITY DEFINER" can be used during creation of function in such language to allow ordinary users to use it.

Other flavour (usually with name that does not end with "u", for example "plpython") can be used by any user. But as it is restricted language, it does not contain dangerous function for accessing files, network, etc.

This distinction between secure and insecure languages allows for writing functions in many languages, but with providing security precautions.

PostgreSQL allows for writing own language. New language must be provided to database as library (*.so or *.dll file) that have one function responsible for running functions written in this language. Library can contain function for checking whether function is correct, but this is not necessary. Creation of programming language are described in Chapter 36 of PostgreSQL documentation (PostgreSQL 2006).

Other commercial databases came with ability to write stored procedures in different languages, but it is not so well thought as in PostgreSQL.

In Microsoft SQL Server 2005 user can write stored procedures in .NET languages. But it is not full .NET suite, as some of classes regarded as dangerous are removed. So this is similar to "safe" languages from PostgreSQL. There is no unsafe languages, and there is no possibility of using languages in two modes - safe and unsafe. Also, there is no possibility of using language that has no .NET implementation.

Oracle has ability to create stored procedures written in Java. But as this is Oracle's flavour of Java, some functions are removed, similarly to Microsoft SQL Server,

System description

- Database contains description of audio and video files
- Client chooses file to receive
- Database server sends streams to the client
- Client receives and displays stream
- Server and client can be on different machines

4.2. Implementation details

Using PostgreSQL as GStreamer source would require implementing new source plugin. Usually all elements of pipeline work inside one process. But PostgreSQL, just like other database servers, is another process, working as other user, very often on different machine. So this is nor practical solution.

GStreamer allows to transfer multimedia stream through the network by using tcpclient and tcpserver plugins. But as not only data but also events and metadata must be transmitted (each pipeline has two buses, data and event one), those plugins require using format that is capable of storing many streams. OGG format is container format that can store many different streams. It can store audio and video streams, together with text and metadata ones. It is standard described in RFC 3533 (Pfeiffer 2003), with available implementations, and is supported by GStreamer.

Instead of creating PostgreSQL source, networking capabilities of GStreamer were used. Server pipeline was build inside PostgreSQL, and client pipeline in application. Thanks to network stream environments and systems in which server and client run are irrelevant, as long as TCP connection is possible between them.

Multimedia files were stored on disk drive. To avoid problems with different file types, playbin was used as source, so it created decoding plugins when needed. In database very simple table, containing only path, human-readable name of multimedia object, and object identifier (integer number) was created, and populated with files on disk.

Fragments of client and server pipelines responsible for transmitting video streams are shown in Figure 3.

SQL functions

get_stream_info(filename) returns information about all streams contained in file

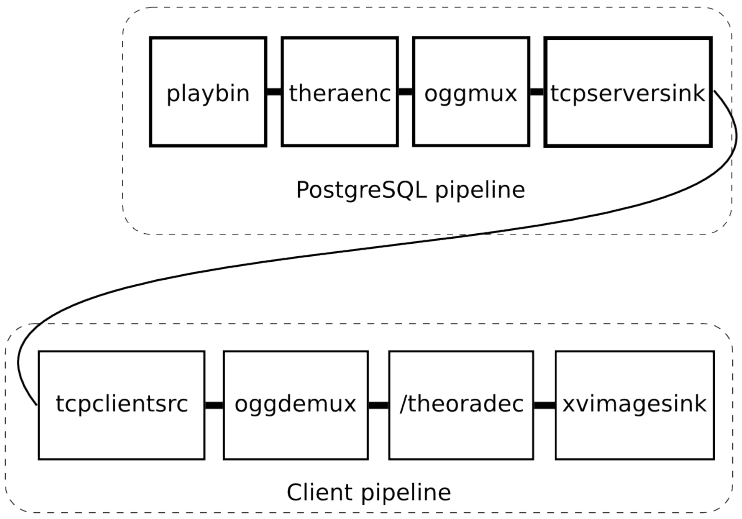

Figure 3. Implemented video pipelines

get_stream(filename, host, port, audio, video) creates pipeline returning requested audio and video stream on requested address and port

Two functions were implemented, both in Python. First one, `get_stream_info`, gets name of file and returns information about it, and information about all streams stored in that file.

Second function, `get_stream`, as input parameters takes name of stream, network address and port to which listen to, and identifier of audio and video streams to transmit. Then it creates network server to which client can connect to and get multimedia stream.

Function creates query plan, and stores it in StaticDictionary, special dictionary used to store data between function calls. There is also GlobalDictionary, shared by all functions in particular session sessions, but as each function can have different plan, it is currently not used. Thanks to storing query plan, it does not need to be computed each time, so resulting stream will be server faster.

To access database function uses library plpy. This library is used to prepare query plan, put parameters (here name) into this plan, and get first row of the result. As only one file is served, only first returned row is used.

Each client requires opening port. This is similar solution to those used in FTP server (Postel 1985), which also uses two ports, one for transmitting commands (here SQL) and one for transmitting data (here multimedia stream).

Only one server can listen on one port. One tcpserver, listening on one port, can send only one stream, so there is no possibility to serve results of different queries on one port. Because each client gives own address and port number server should listen to, it is possible to serve to many clients concurrently.

Inside function `get_stream` events generated by GStreamer are inspected by message loop. The only duty of message handler is to stop loop (so end entire function) when end of stream is reached or client has disconnected. So function stops (by calling loop.quit()) and further processing can occur when stream has ended. But if function quits when stream has ended, and client calling it wants to process this stream, there can be problem with deadlock. To be able to process stream sent by database, client program may put query in another thread, or just make asynchronous call, and gather result after finishing playing.

4.3. Problems and limitations

GStreamer is still actively developed. It is used in many programs, is even foundation of GNOME Desktop Environment multimedia system, but still has not reached status of full stability. It's current version is 0.10, and it can be used in programs, but it's API and ABI can be changed. Due to this fact, there can still be some problems. During implementation of system problems with cooperating GNonLin and network stream were encountered, so GNonLin was not used. Instead playbin was used as source of multimedia data. So now there is no possibility to join many streams, and to play streams with non-standard speed.

Dividing transmission between two ports has advantages and disadvantages. Having many open ports is disadvantage, as it may require additional configuration of firewall. Also, when PostgreSQL allows for limiting access to different databases basing on address, port, user name, etc., it is not possible to limit access to opened multimedia port in the same way.

But using different ports for different data types means that we can use existing solutions without worrying how to allow transmitting multimedia stream without spoiling SQL client. Also, it is possible to give more priority to port used to transmit stream, as it is extremely important that such data has highest priority to be transferred without glitches. So, different ports allow for making QoS (Quality Of Service) policy. SQL port can have lower priority, as it transmit less data, and data not so sensitive to delays as multimedia stream.

Currently at most one audio and one video stream can be transmitted. As OGG format supports storing many streams inside, more than two streams can be transmitted concurrently. But such possibility will complicate pipelines on the server and on the client, and will require additional logic to decide how to process each of the transmitted streams.

5. Conclusion

Article shows possibility of storing multimedia data in database and serving them from database. Such database becomes central point of multimedia system. Thanks to using existing components code that need to be written is not complicated. Also, when using existing, standard solutions, programmers familiar with them will be able to understand, use and enhance system.

Article presents work in progress, and many ideas are yet to be implemented.

One possibility of enhancing system is to treat multimedia files as special data types. Custom types can have GIN or GiST indexes (Chapters 50 and 51 of PostgreSQL 2006), which enables faster queries taking special attributes into consideration. Such multimedia types can have own operators; some advanced operators provided for geometric types (Chapter 9.10 of PostgreSQL 2006), can serve as example. Additional operators (implemented as GStreamer plugins) could be used to extract attributes describing streams. Aggregate operators would allow to treat multimedia file as array or table, and to operate on single frames without much trouble. And when access to single frames is provided, multimedia file can be treated as temporal data. Using temporal models to manage multimedia data can open many new possibilities.

But the largest area of possible development are methods of querying. Currently function is used to ask for data, but when special operator are present it could be possible to write ordinary SQL query. But such query would return entire stream. Limiting stream to fragment (for example by using OFFSET 15:00 LIMIT 23s) would require rewriting query, and this can require changing query processing mechanism. Althoung it is possible to access internals of PostgreSQL server by using Server Programming Interface (described in Chapter 41 of PostgreSQL 2006), but it is rather complicated task. And it may be probable that such deep intervention will not be possible by using existing mechanisms; also such new changed SQL could be incompatible with other databases and tools.

References

Alty, J. (2004) *Multimedia.* In Allen B. Tucker, editor, *Computer Science Handbook.* Chapmann & Hall/CRC, New York.

Czerwinski, M. and Gage, D. and Gemmell, J. and Marshall, C. et. al. (2006) *Digital memories in an era of ubiquitous computing and abundant storage.* In Communications of ACM 49, 1, 44–50.

Fonseca, R. and Almeida, V. and Crovella, M. (2005) *Locality in a web of streams.* In Communications of ACM 48, 1, 82–88.

Gamma, E. and Helm, R. and Johnson, R. and Vlissides, J. (1994) *Design Patterns. Elements of Reusable Object–Oriented Software.* Addision–Wesley, New York.

Gemmell, J. and Bell, G. and Lueder, R. (2006) *MyLifeBits: a personal database for everything.* In Communications of ACM 49, 1, 88–95.

The PostgreSQL Global Development Group (2006) *PostgreSQL 8.2.4 Documentation.* http://www.postgresql.org/docs/8.2/static/index.html

Kent, W. (1983) *A simple guide to five normal forms in relational database theory.* In Communications of ACM 26, 2, 120–125.

Oria, V. and Li, Y. and Dorai, C. (2004) *Multimedia Databases: Analysis, Modeling, Quering, and Indexing.* In Allen B. Tucker, editor, *Computer Science Handbook.* Chapmann & Hall/CRC, New York.

Pfeiffer, S. (2003) *The Ogg Encapsulation Format Version 0.* http://www.faqs.org/rfcs/rfc3533.html

Postel, J. and Reynolds, J. (1985) *File Transfer Protocol (FTP).* http://www.faqs.org/rfcs/rfc959.html

Pratt, W. and Unruh, K. and Civan, A. and Skeels, M. (2006) *Personal health information management.* In Communications of ACM 49, 1, 51–55.

Rejaie, R. (2006) *Anyone can broadcast video over the internet.* In Communications of ACM 49, 11, 55–57.

Silbershatz, A. and Korth, H. and Sudarshan, S. (2004) *Data models.* In Allen B. Tucker, editor, *Computer Science Handbook.* Chapmann & Hall/CRC, New York.

This work was supported by the grant S/WI/2/03 from the Bialystok Technical University.