
ORACLE®

ORACLE

An API for Reading the MySQL Binary Log

Mats Kindahl
Lead Software Engineer, MySQL Replication & Utilities

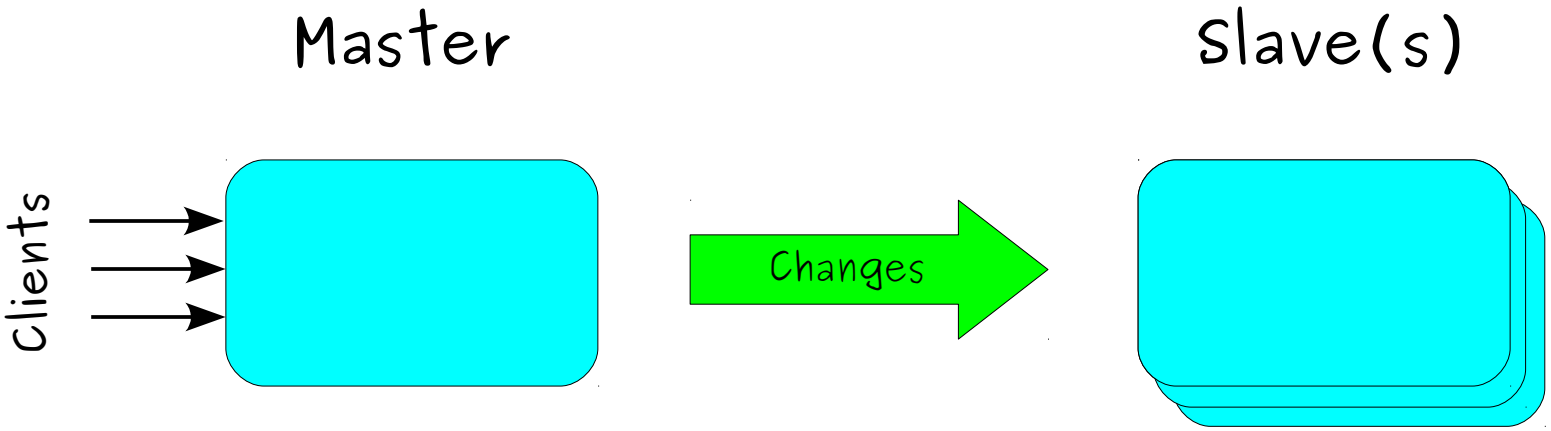
Lars Thalmann
Development Director, MySQL Replication, Backup & Connectors

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.
The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

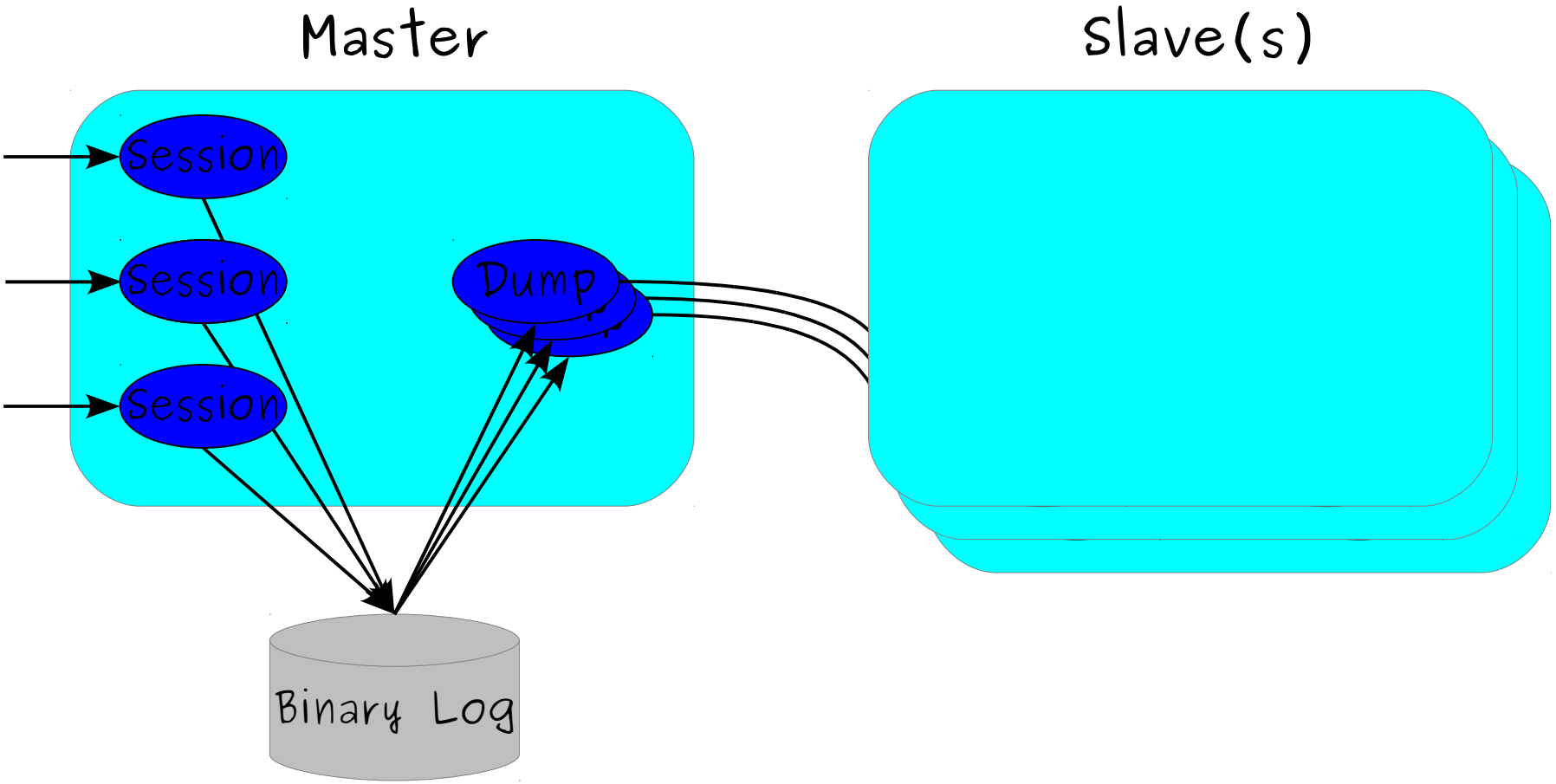
Outline

- Replication Architecture
- Binary logs
- Binary log event
- Reading binary log
  - Connecting to server
  - Reading from files
- Reading events
  - Queries
  - Reading rows (row-based replication)
  - Other events

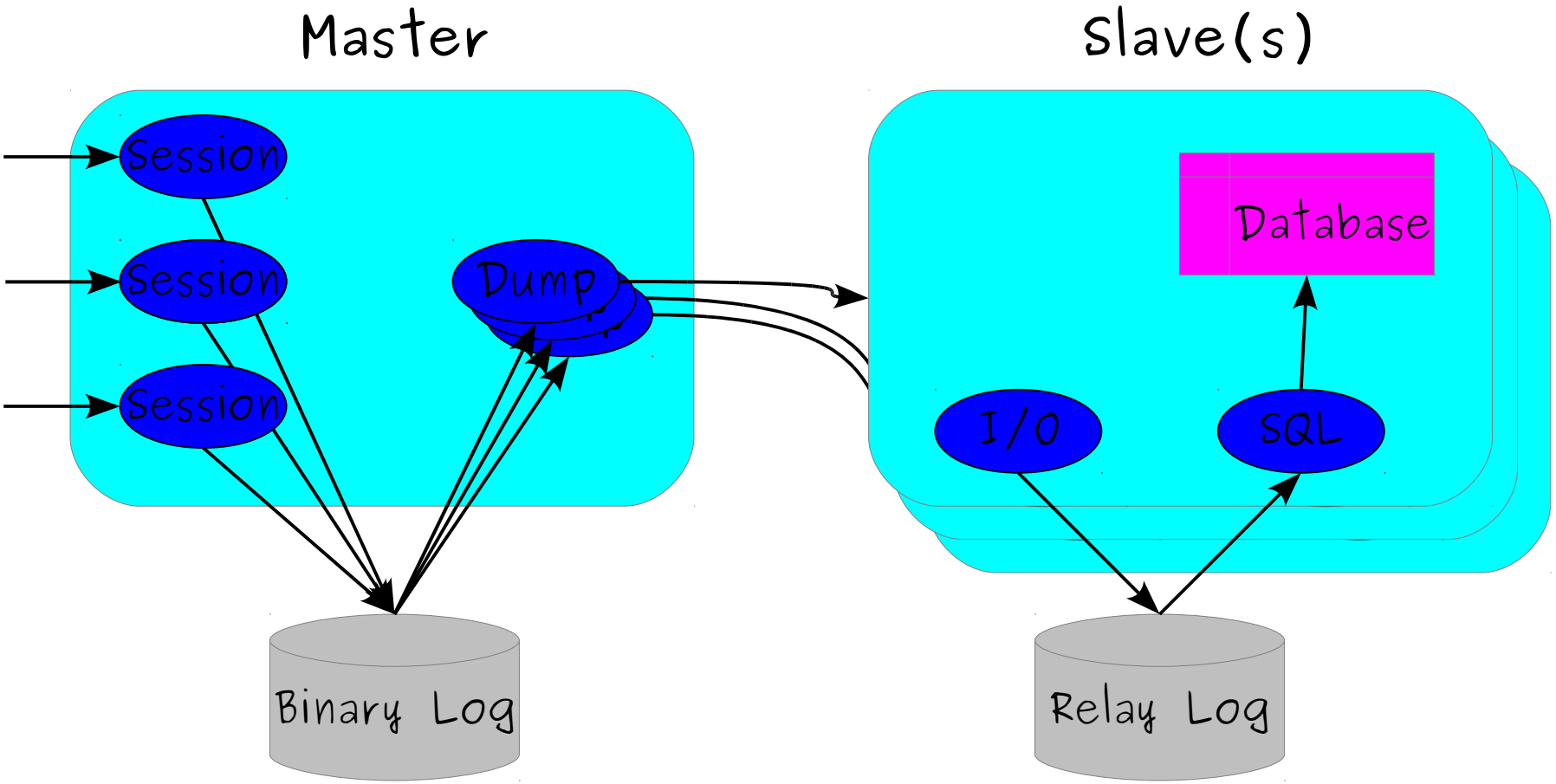
Replication Architecture



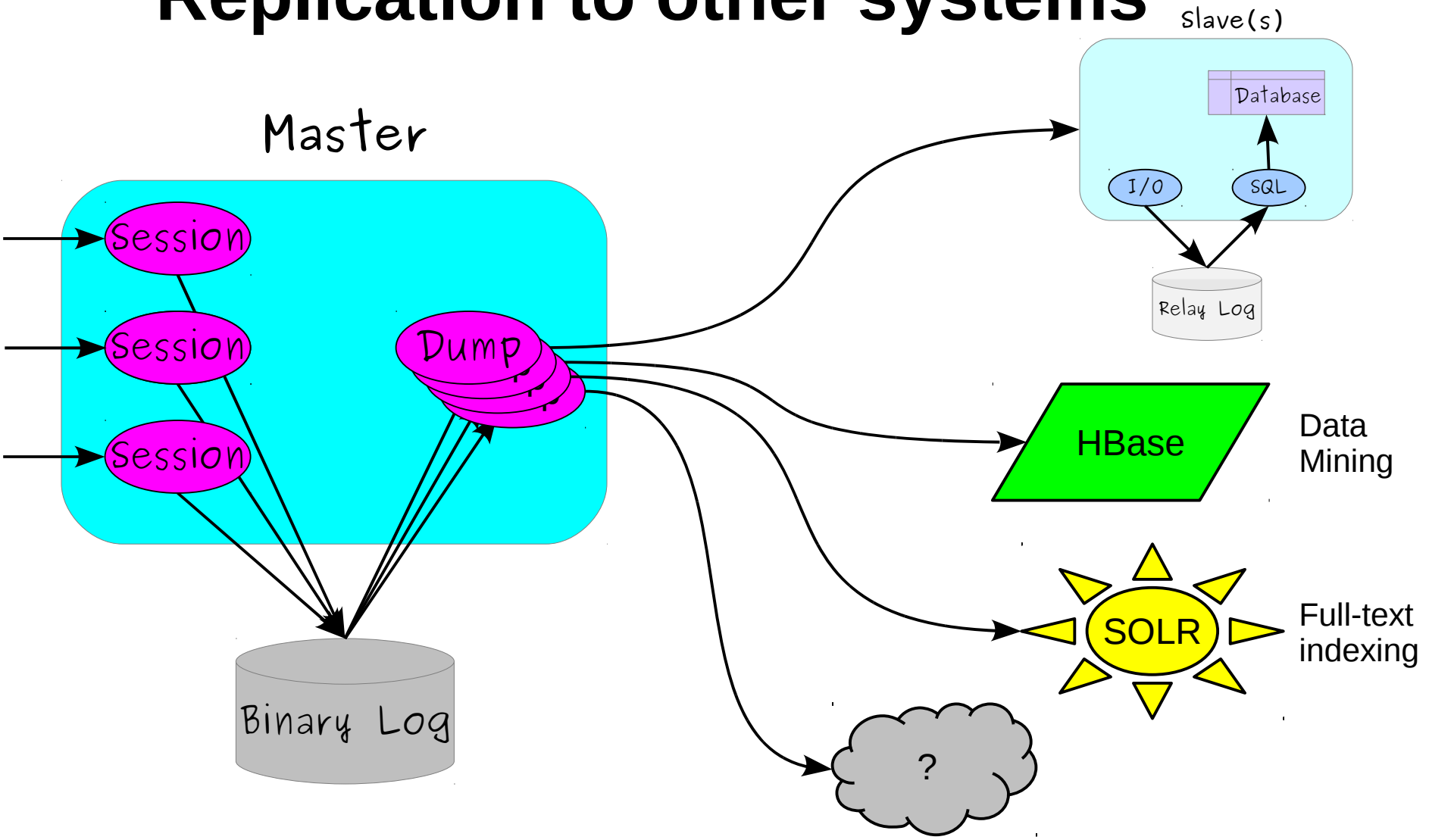
Replication Architecture



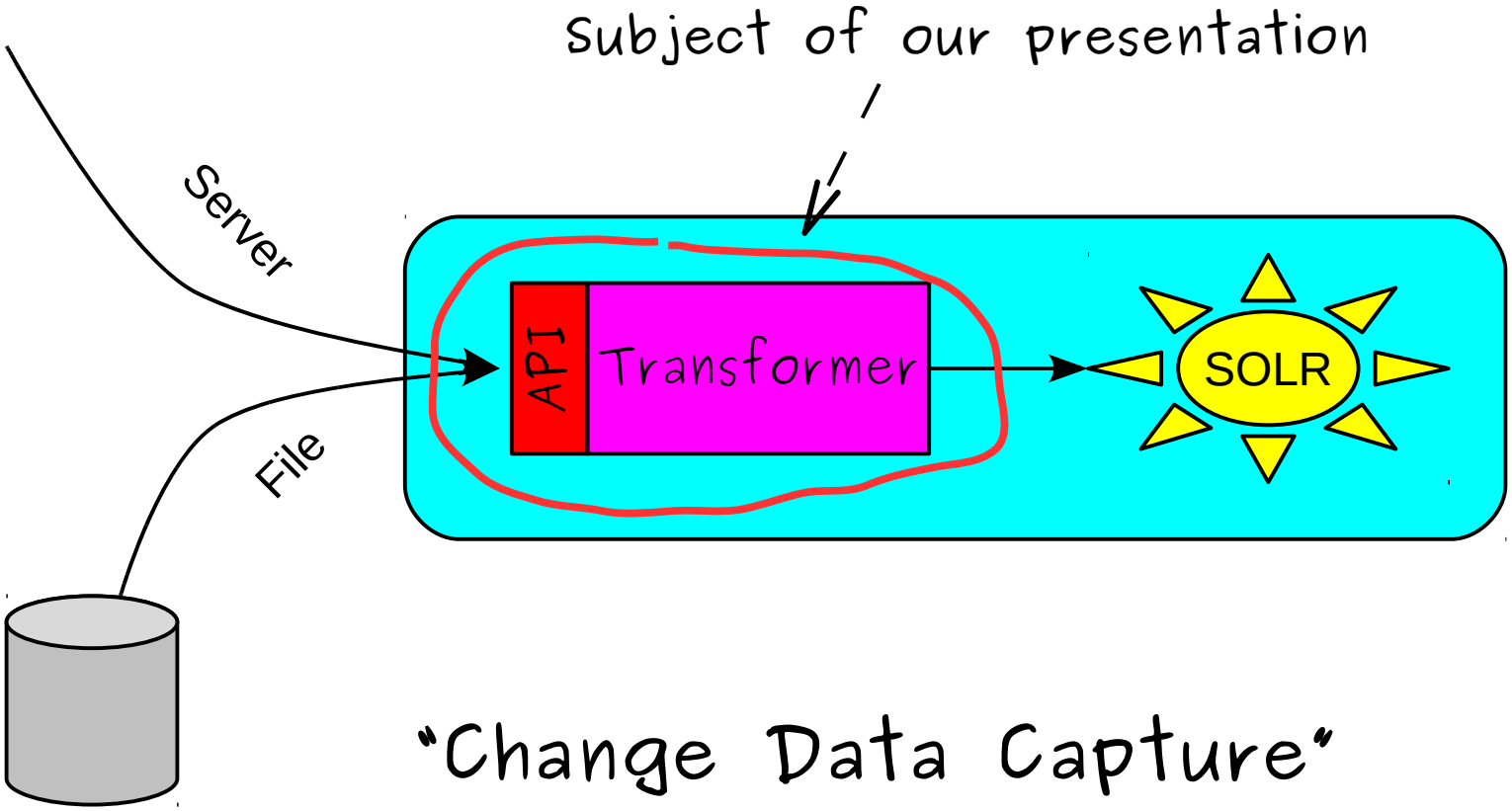
Replication Architecture



Replication to other systems



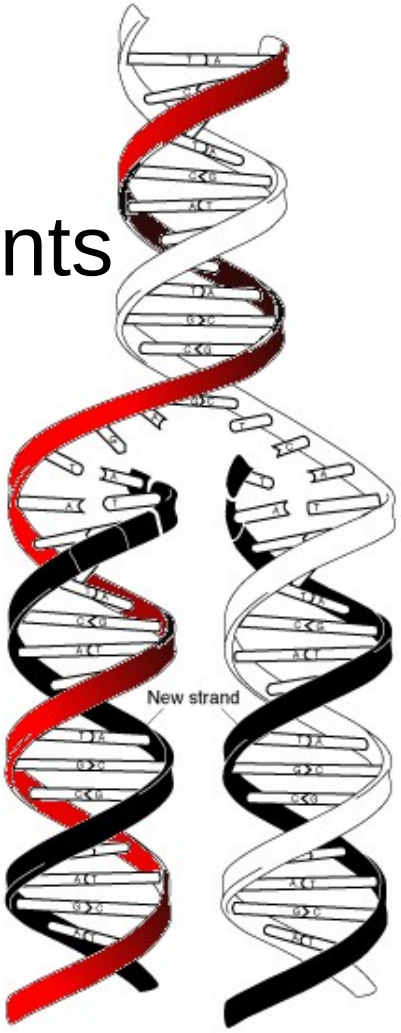
Transforming events



"Change Data Capture"

Binlog API

- Library to process replication events
- API is ready for use
- Goals:
  - Simple
  - Extensible
  - Efficient



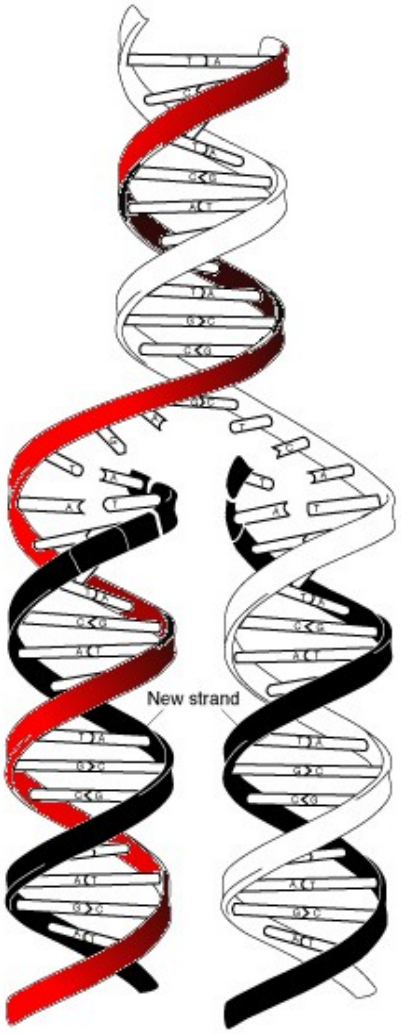
Binlog API

- The replication listener

How to capture events

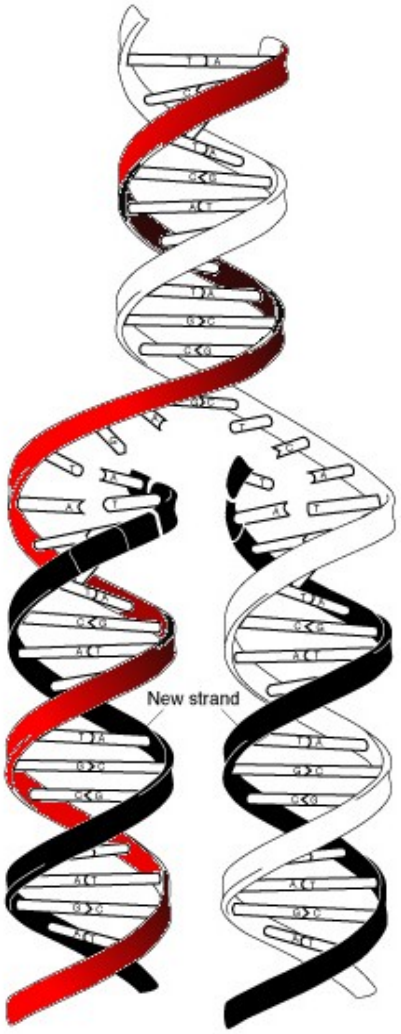
First example

```
#include <cstdlib>
#include <iostream>
#include <binlog_api.h>

int main(int argc, char *argv[]) {
  const char *url = “mysql://root@127.0.0.1:3360”;
  Binary_log binlog(create_transport(url));
  binlog.connect();
  Binary_log_event *event;
  while (true) {
    int result = binlog.wait_for_next_event(&event);
    if (result == ERR_EOF)
      break;
    cout << “ at “ << binlog.get_position()
         << “ event type “ << event.get_type_code()
         << endl;
  }
  return EXIT_SUCCESS;
}
```



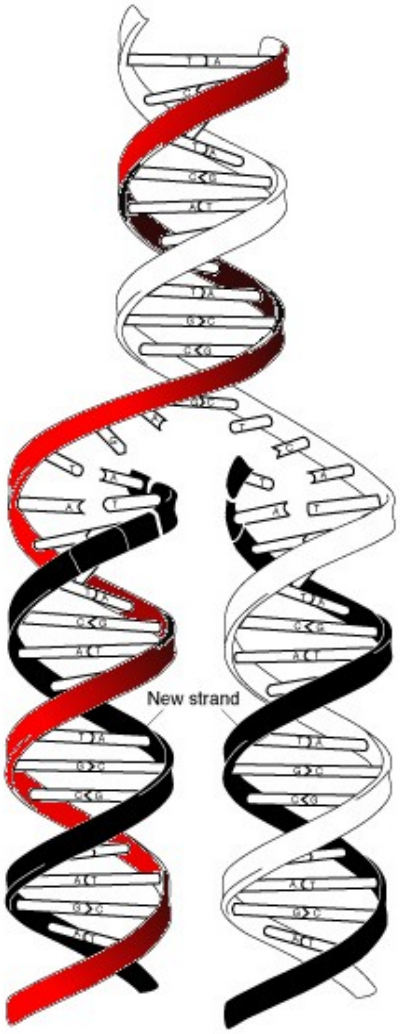
Create network transport

```
#include <cstdlib>
#include <iostream>
#include <binlog_api.h>

int main(int argc, char *argv[]) {
  const char *url = "mysql://root@127.0.0.1:3360";
  Binary_log binlog(create_transport(url));
  binlog.connect();
  Binary_log_event *event;
  while (true) {
    int result = binlog.wait_for_next_event(&event);
    if (result == ERR_EOF)
      break;
    cout << " at " << binlog.get_position()
         << " event type " << event.get_type_code()
         << endl;
  }
  return EXIT_SUCCESS;
}
```



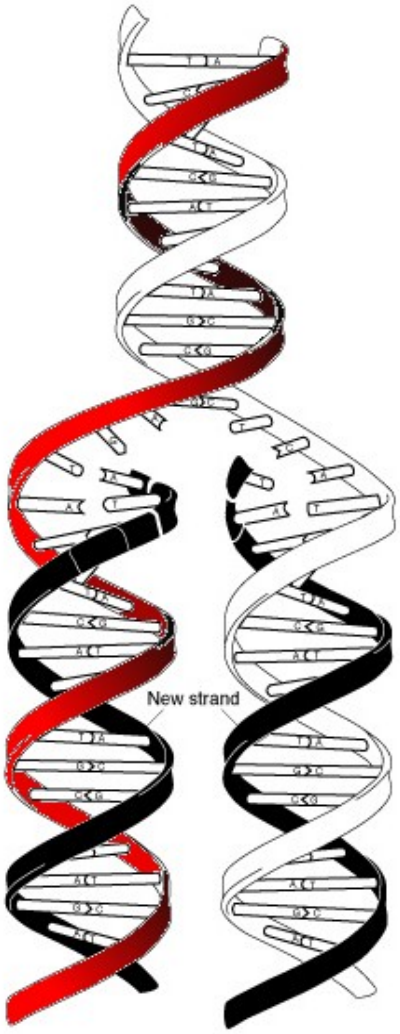
… or file transport

```
#include <cstdlib>
#include <iostream>
#include <binlog_api.h>

int main(int argc, char *argv[]) {
  const char *url = "file:///tmp/binlog.0000001";
  Binary_log binlog(create_transport(url));
  binlog.connect();
  Binary_log_event *event;
  while (true) {
    int result = binlog.wait_for_next_event(&event);
    if (result == ERR_EOF)
      break;
    cout << " at " << binlog.get_position()
         << " event type " << event.get_type_code()
         << endl;
  }
  return EXIT_SUCCESS;
}
```



Connect the transport

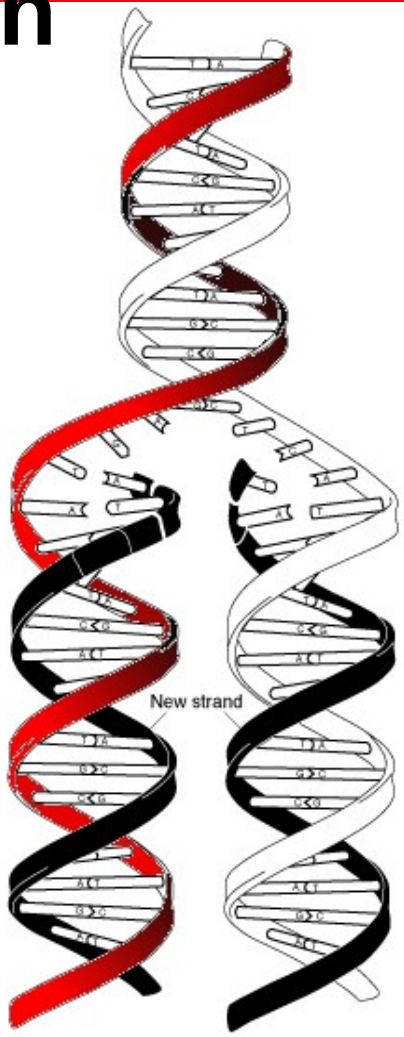
```
#include <cstdlib>
#include <iostream>
#include <binlog_api.h>

int main(int argc, char *argv[]) {
  const char *url = "file:///tmp/binlog.0000001";
  Binary_log binlog(create_transport(url));
  binlog.connect();
  Binary_log_event *event;
  while (true) {
    int result = binlog.wait_for_next_event(&event);
    if (result == ERR_EOF)
      break;
    cout << " at " << binlog.get_position()
         << " event type " << event.get_type_code()
         << endl;
  }
  return EXIT_SUCCESS;
}
```



Digression: set read position

- Default: start at beginning
- Set position explicitly:

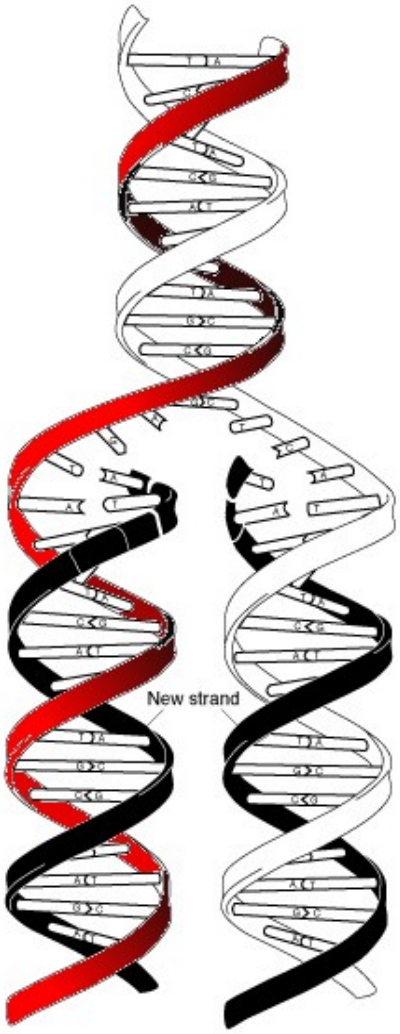
```
if (binlog.set_position(file, pos))
{
  /* Handle error */
}
```



Read events

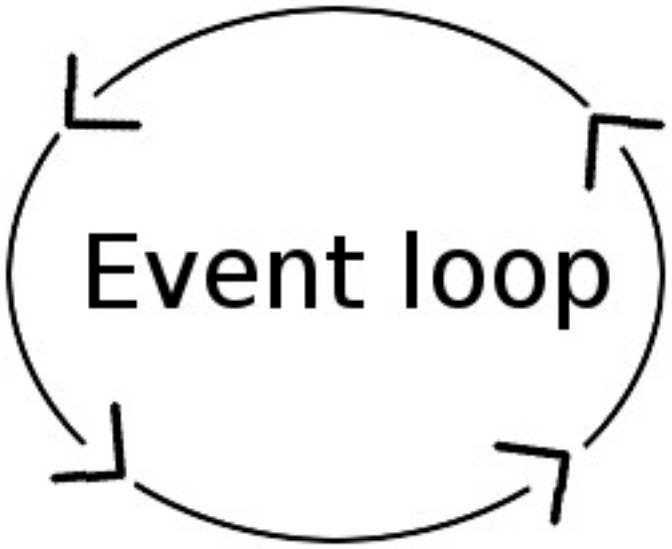
```
#include <cstdlib>
#include <iostream>
#include <binlog_api.h>

int main(int argc, char *argv[]) {
  const char *url = "file:///tmp/binlog.0000001";
  Binary_log binlog(create_transport(url));
  binlog.connect();
  Binary_log_event *event;
  while (true) {
    int result = binlog.wait_for_next_event(&event);
    if (result == ERR_EOF)
      break;
    cout << " at " << binlog.get_position()
         << " event type " << event->get_type_code()
         << endl;
  }
  return EXIT_SUCCESS;
}
```

Get event



Steps summary

- Create a transport
  - `create_transport`
- Connect to server
  - `connect`
- Set position
  - `set_position`
- Start event loop
  - `wait_for_next_event`

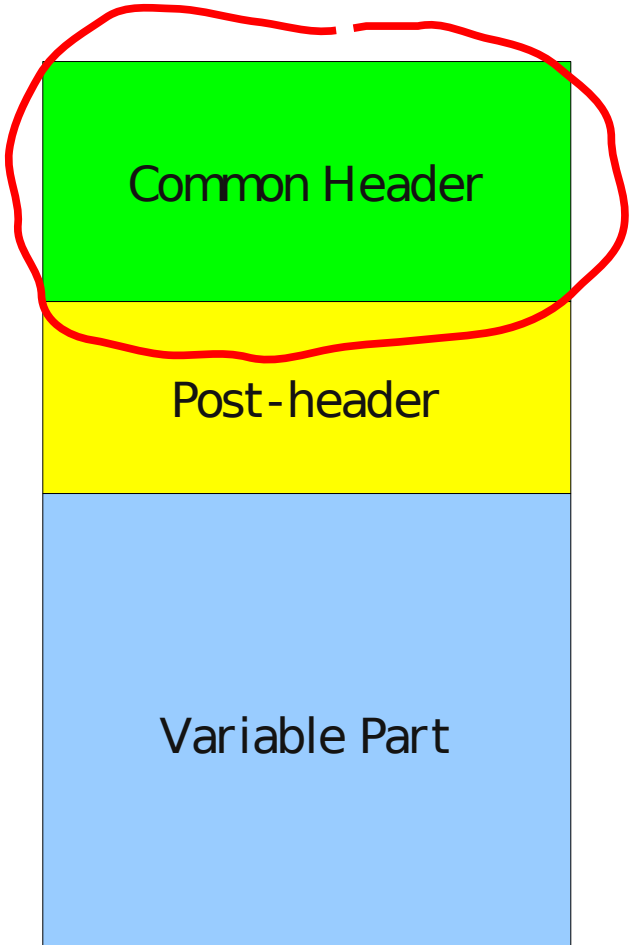


Binlog API

- The replication listener

Reading information in events

Binlog Event Structure

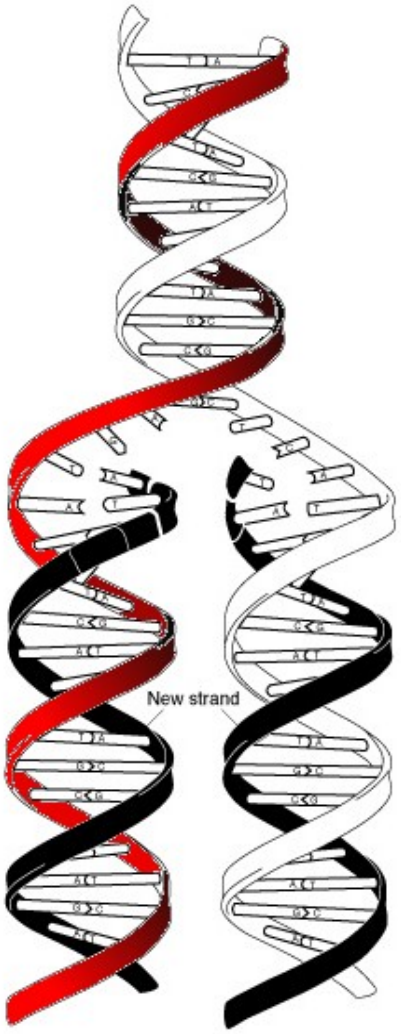


- Common header
  - Generic data
  - Fixed size
- Post-header
  - Event-specific data
  - Fixed size
- Variable part
  - Event-specific data
  - Variable size

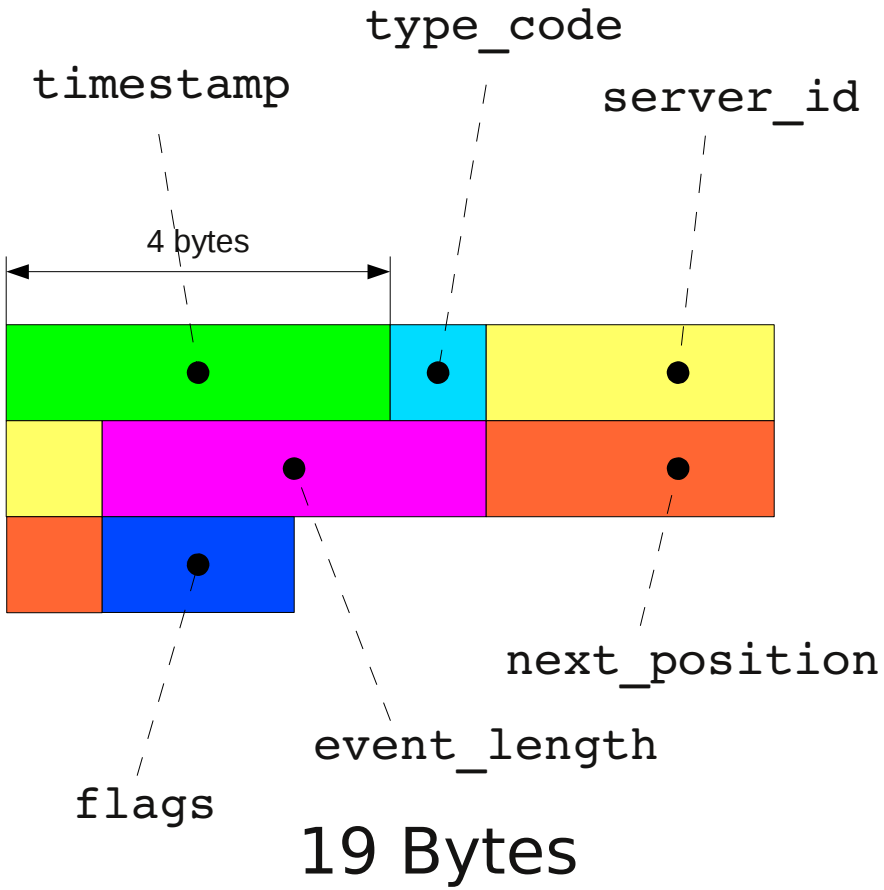
Reading the header

- Read common header
  - `header()`
- Access fields

```
switch (event->header()->type_code) {
case QUERY_EVENT:
  …
case USER_VAR_EVENT:
  …
case FORMAT_DESCRIPTION_EVENT:
  …
}
```

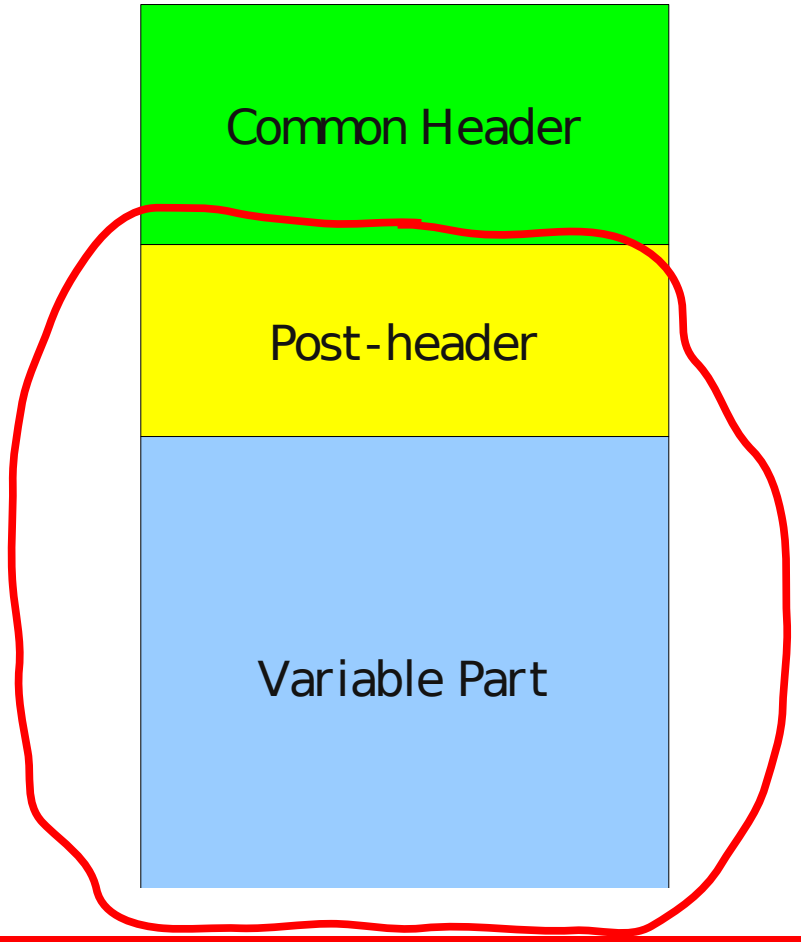


Binlog Event Common Header



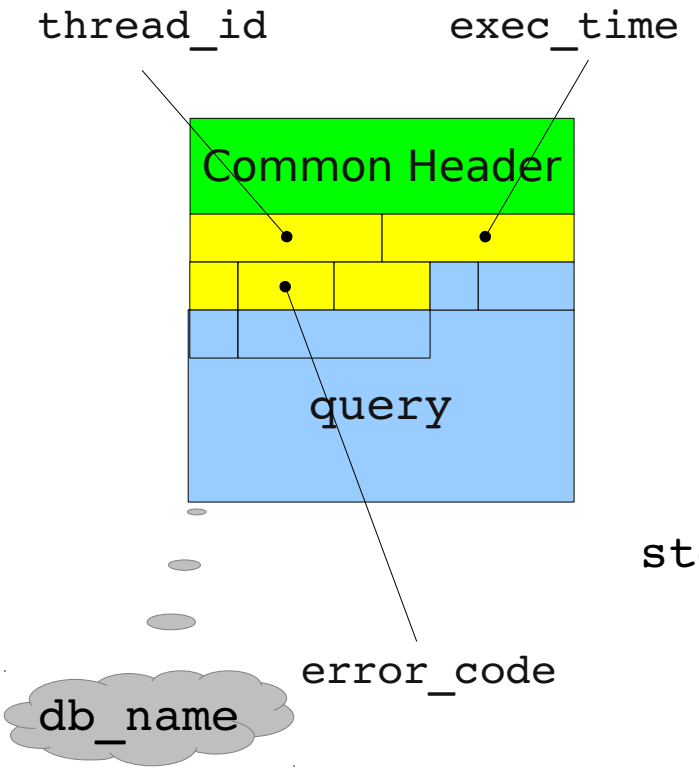
- Data common to all events
- Next Position
  - *One-after-end* of event
- Timestamp
  - Statement *start* time
- Flags
  - Binlog-in-use
  - Thread-specific
  - Suppress “use”
  - Artificial
  - Relay-log event

Binlog Event Structure



- Common header
  - Generic data
  - Fixed size
- Post-header
  - Event-specific data
  - Fixed size
- Variable part
  - Event-specific data
  - Variable size

Query Event



- Most common event
- Used for statements
- Statement logged literally
  - … in almost all cases

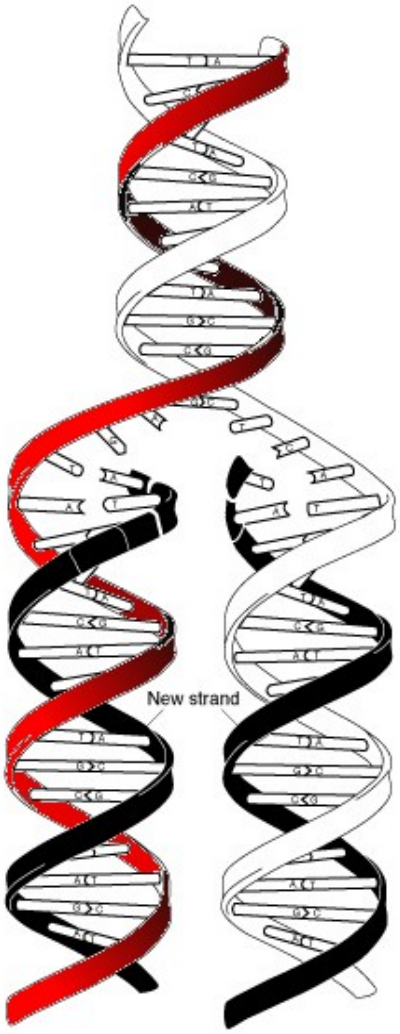
```
std::vector<uint8_t> variables
```

Special case: need to be decoded

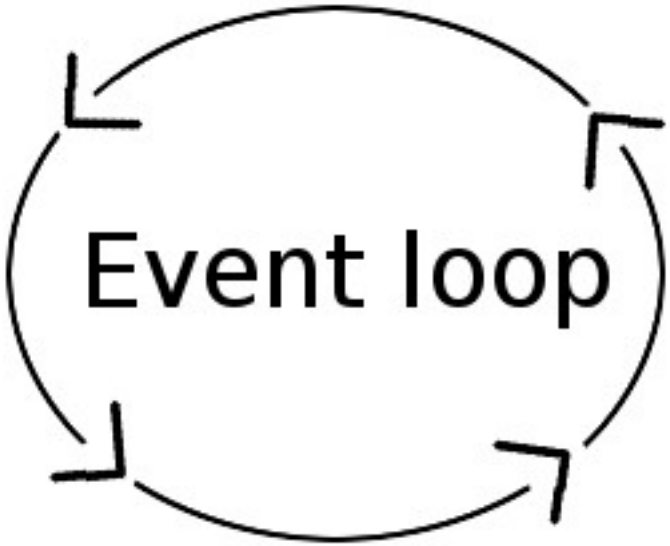
Reading event data

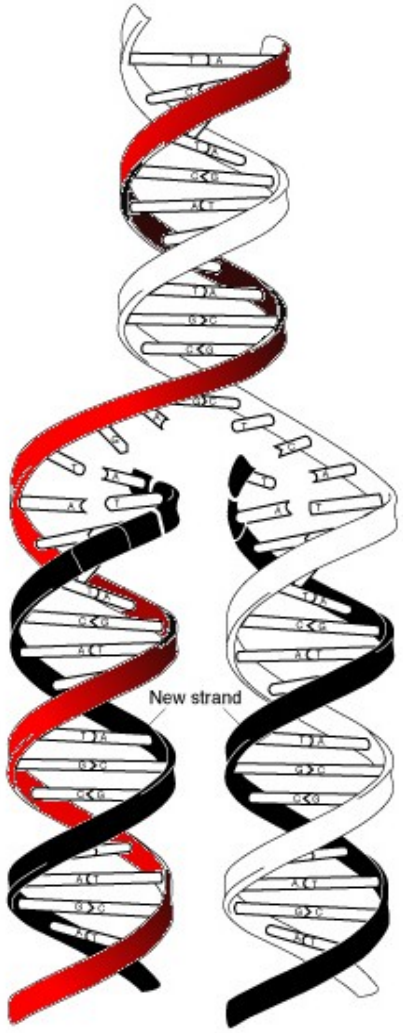
- Cast to correct event type
- Access fields

```
switch (event->header()->type_code) {
case QUERY_EVENT:
  Query_event *qev =
    static_cast<Query_event*>(event);
  cout << qev->query << endl;
  break;
case USER_VAR_EVENT:
  …
case FORMAT_DESCRIPTION_EVENT:
  …
}
```



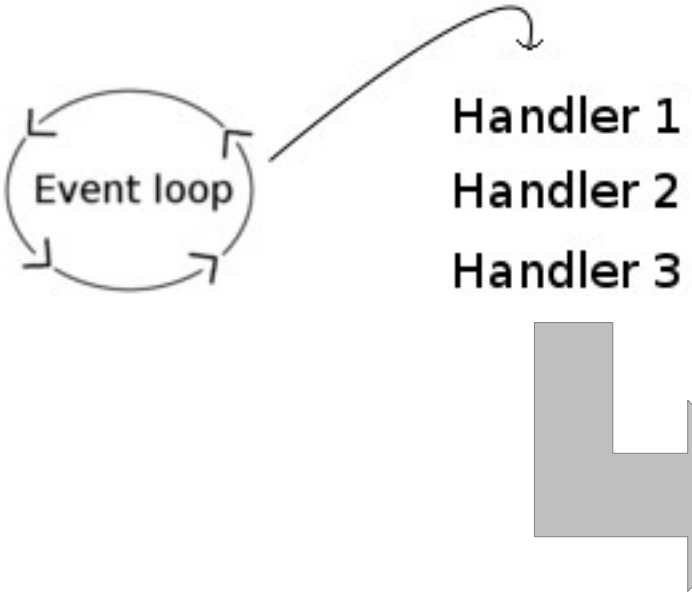
Event-driven API

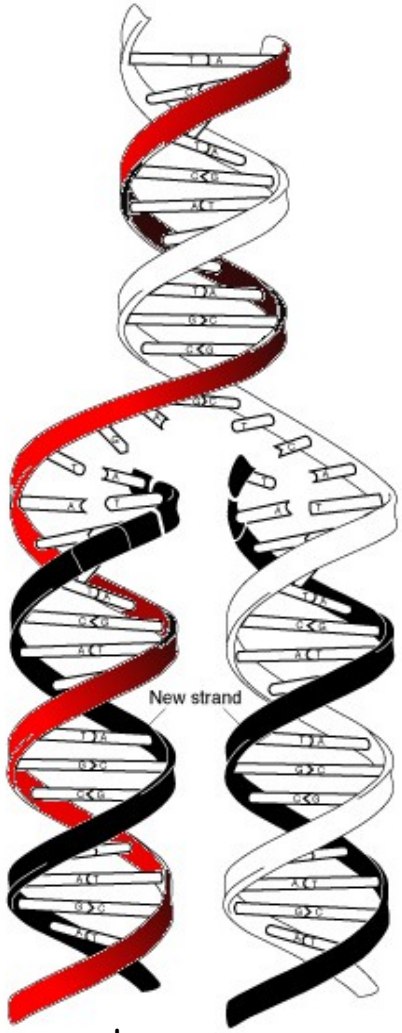




Event-driven API

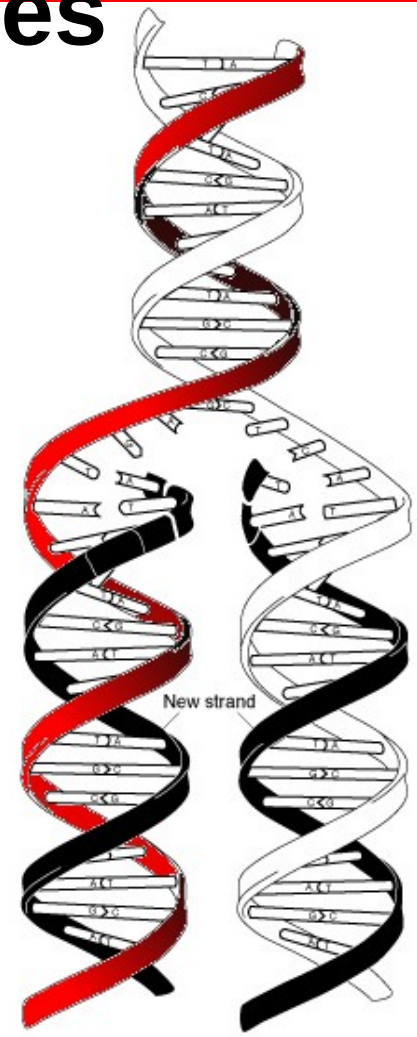
- Content handlers





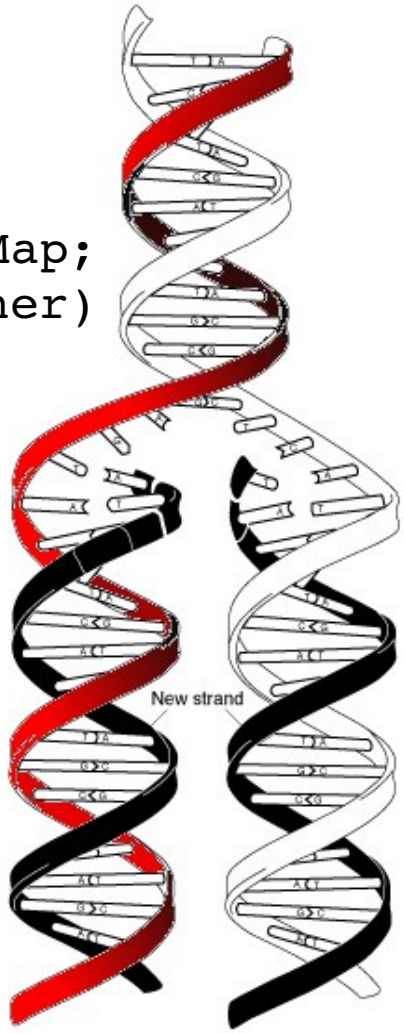
Saving user-defined variables

```
class Save_handler
  : public Content_handler
{ … };

Save_handler::Map vars;
Save_handler save_vars(vars);
binlog.content_handler_pipeline()
  ->push_back(&save_vars);
```



User-defined variables

```
class Save_handler : public Content_handler {
public:
  typedef std::map<std::string, std::string> Map;
  Save_handler(Map &container) : m_var(container)
  { }

  Binary_log_event *
  process_event(User_var_event *event) {
    m_var[event->name] = event->value;
    return NULL;
  }

private:
  Map &m_var;
};
```



Replace handler

```
class Replace_vars :
  public Content_handler
{
  Binary_log_event *
  process_event(Query_log_event *event)
  {
    /* Code to replace variables */
  }
};
```

Full example: basic-2.cpp

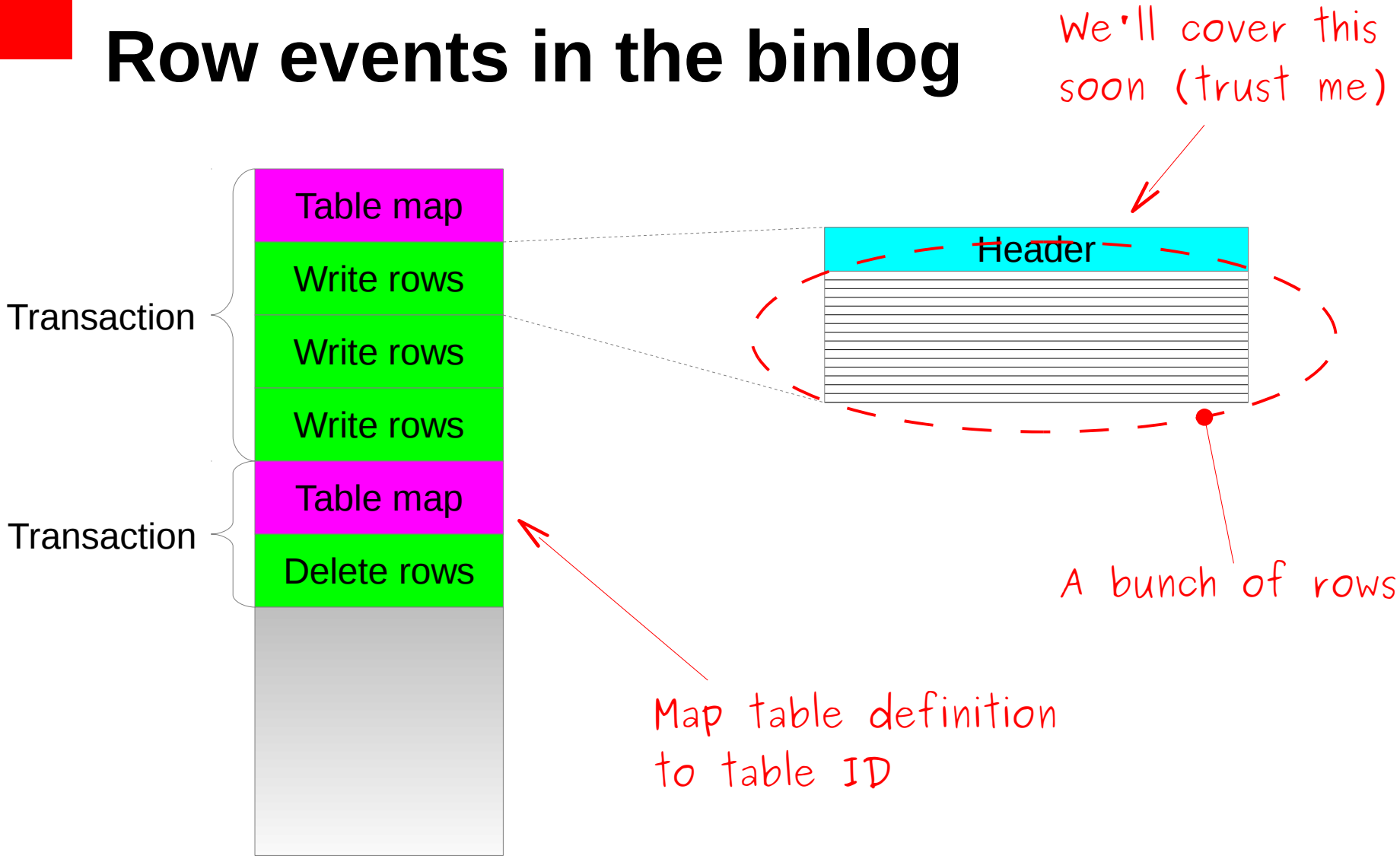
Binlog API

- The replication listener

Example two:

How to capture live row changes

Row events in the binlog



Capturing row events

```
class Row_event_handler :
  public Content_handler
{
public:
  Binary_log_event *
  process_event(Row_event *event)
  {
    switch(ev->header()->type_code)
    {
      case WRITE_ROWS_EVENT:
      case UPDATE_ROWS_EVENT:
      case DELETE_ROWS_EVENT:
        ...
```

Capturing row events

- The *_ROWS_EVENT

Defined in the table map event

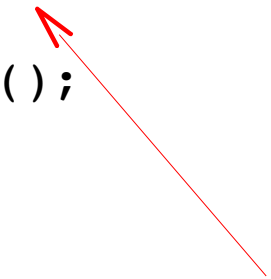
```
uint64_t table_id;
uint16_t flags;
uint64_t columns_len;
uint32_t null_bits_len;
vector<uint8_t> columns_before_image;
vector<uint8_t> used_columns;
vector<uint8_t> row;
```

Raw row data

Reading rows

- Wrap raw row data in **Row_event_set**
- Iterate over rows using iterator

```
Row_event_set rows(row_event, table_map_event);

Row_event_set::iterator it= rows.begin();
```



You need to have captured this before!

Reading fields of a row

- **Row_of_fields** to iterate fields of a row
  - Turns *row* into *row of fields* sequence

```
Row_event_set rows(row_event, table_map_event);

for (Row_event_set::iterator it = rows.begin() ;
     it != rows.end() ;
     ++it)
  table_delete(os.str(), Row_of_fields(*it));
```

Reading fields of a row

- Iterate over fields in Row_of_fields

```
void table_delete (..., const Row_of_fields& fields)
{
 Row_of_fields::iterator it= fields.begin();
 for (int id = 0 ; it =! fields.end() ; ++it, ++id) {
  std::string str;
  Converter().to(str, *it);
  std::cout << id << "= " << str << std::endl;
 }
}
```

Decoding a field

- Iterate over fields in Row_of_fields

```
void table_delete (..., const Row_of_fields& fields)
{
 Row_of_fields::iterator it= fields.begin();
 for (int id = 0 ; it =! fields.end() ; ++it, ++id) {
  std::string str;
  Converter().to(str, *it);
   std::cout << id << "= " << str << std::endl;
 }
}
```

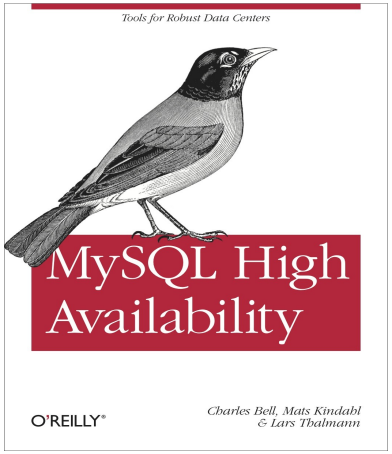
Summary – what's it for?

- Replicate to other systems
  - Hbase, SOLR, etc.
- Triggering on specific events
  - Call the DBA when tables are dropped?
  - Monitor objects in database
- Browsing binary logs
  - Extract subset of changes (by table, by execution time, etc.)
  - Statistics
- Component in building other solutions
  - Point-in-time restore
  - Sharding / Load-balancing

Summary – what we've covered

- Reading events
- Creating content handlers
- Processing queries
- Processing rows
- Reading fields
- … but there is a lot more

- **Available at labs**
  - **http://labs.mysql.com/**
- **Source code available at launchpad**
  - **http://launchpad.net/mysql-replication-listener**
- **MySQL High Availability**
  Get it as free ebook: http://oreilly.com/go/ebookrequest
  Valid this week, mention event "MySQL Replication Update"




ORACLE®