

JOINT INSTITUTE FOR NUCLEAR RESEARCH

[Dzhelepov](https://web.archive.org/web/20031204011713/http%3A/dzhelepov.jinr.ru/) laboratory of Nuclear Problems

**FINAL REPORT ON THE**

**START PROGRAMME**

*SPD EventIndex*

**Supervisor:**

Fedor Valerievich Prokoshin

**Student:**

Zamira Budtueva, Russia
North Ossetian State University

**Participation period:**

July 16 – September 09,

Summer Session 2023

Dubna, 2023

TABLE OF CONTENTS

Abstract 3

Introduction 4

History of development 4

SPD information ecosystem 5

Data generation 9

Optimization of data recording speed 9

Conclusion 16

Bibliography 17

Acknowledgments 18

АBSTRACT

SPD EventIndex is a catalog of physical events received from the SPD detector at NICA collider or modeled for data analysis and processing. EventIndex should provide indexing of event data, transmission of this information and writing to the database, access to it by users through interactive and asynchronous interfaces, as well as by data processing and analysis programs through the API.
The development of the EventIndex begins with the server part of the repository, at the moment it is a PostgreSQL DBMS. A simple interface was created: a command-line client and a graphical web interface for event information retrieval. A messaging service is being developed for asynchronous processing of requests for large amounts of data.

Currently, research is underway to improve the speed of loading data into the storage. Various optimization methods are being studied, and programs are being tested with their application. A comparative analysis of various modules for interaction with PostgreSQL is carried out, such as: aiopg, asyncpg, pg8000, psycopg2, PyGreSQL, SQLAlchemy. Results of the tests, used to identify the most effective module, are presented.

Work is also underway to optimize the speed of reading and loading data by changing PostgreSQL parameters, based on the need to increase performance and reliable data security in case of failures.

INTRODUCTION

The SPD experiment is a research project aimed at studying the physics of elementary particles and searching for new fundamental knowledge about the interactions and structure of particles. The main goal of this project is to analyze the data obtained from the experimental SPD detector installed at the Nica particle accelerator, created at the Joint Institute for Nuclear Research, located in the city of Dubna, Moscow region.

The SPD installation is a large-scale system of detectors capable of registering and analyzing various events that occur during collisions of nuclei and elementary particles with high energy.

To store and process a large amount of data obtained during the SPD experiment, it will be necessary to create complex information systems. One of such systems should be an Event Index - a catalog of physical events received from the installation or modeled for data analysis and processing.

HISTORY OF DEVELOPMENT

When developing the IS of the SPD experiment, it is assumed to widely use the experience gained in the development of similar systems for existing experiments. In particular, in the ATLAS experiment at the LHC accelerator, there is a similar catalog of events - the ATLAS EventIndex system. Each year, the experiment produces about 30 billion events, and about 100 billion are generated additionally. Event records are stored in files located on nodes of the GRID distributed computing system. The presence of such a huge amount of information determined the need to create a global catalog that would allow determining the location of each record for processing and analysis.

In the SPD experiment, data volumes of the same order as on ATLAS are expected with even more events. Despite the differences in the tasks set in these experiments, their data processing systems have similar features, which makes it possible to use similar solutions. SPD is supposed to use a similar data model and file organization, as well as a distributed data storage and processing system. Groups of statistically equivalent events are stored in files on disk or on magnetic tape. Each file usually contains from 1000 to 10000 events, depending on the format. Files are grouped into datasets, usually containing events related to a single data-taking session (Run). The main difference between the experiments is the absence of a trigger, instead, the initial selection of data will be carried out by an online filter based on machine learning methods.

SPD INFORMATION ECOSYSTEM

In addition to EventIndex, various databases and information systems will be used on the SPD installation:

• Databases of equipment and connection mapping;

• Databases of data-taking conditions and calibrations;

• Distributed computing and data storage management systems;

• Database of physical metadata;

• Monitoring systems;

• Logging and accounting systems;

• Database of personnel and publications.

At this stage of the creation of the SPD experiment, information and computing systems are in the initial stage of development. The development of EventIndex is planned to start with parts that do not depend on other components.

SPD EventIndex is being developed as a comprehensive information system that should provide :

• obtaining information about experimental events and simulated data by indexing data files containing information about these events;

• transfer of this information and write to databases;

• access to information for data processing and analysis programs via API and applications;

• access to information to users through interactive and asynchronous interfaces.

Figure 1 shows a preliminary diagram of the EventIndex architecture.



Figure 1: SPD Event Index data flow architecture and schema

The choice of a platform for data storage and management was made based on the expected flows and volumes of information, the content of the EventIndex record and the expected use cases.

The estimated data flow from the installation will be 20 GB/s, at the output of the online filter we expect a 20-fold reduction in the data flow to 20-23 thousand events per second or approximately 30 billion events per year, while the size of the event record will be significantly smaller than for ATLAS. The absence of the need to store detailed trigger information and references to derived data types allows you to have a relatively small record size in EventIndex and a homogeneous set of fields, permiting use of a conventional relational DBMS instead of the hybrid HBase-based solution used in ATLAS.

An entry for an event in EventIndex must contain the following fields:

• event IDs: Run number (run\_number) and event number in Run (event\_number)

• information about online filter solutions, in the form of a bit mask (olf\_result)

• unique identifier of the RAW data file containing this event (fuid\_raw). Using the UUID of a file, you can access it through a distributed storage system.

• ID of the dataset this file is included in (dsid\_raw)

As the data is processed, new instances of the recovered events will be created in a format optimized for physical analysis (AOD). Pointers to different versions of such files will be added to the event record. Also, important event parameters can be added to the record, which can be used for classification and selection.

The PostgreSQL database was chosen for reliable storage and processing of structured data. It is widely used for information management in various applications, from small websites to large corporate systems. PostgreSQL supports SQL standards and provides a rich set of functions for working with data, including support for complex queries, transactions, indexes, views and stored procedures. One of the key features of PostgreSQL is its ability to process large amounts of data and support multithreading, which makes it an excellent choice for applications with high performance and reliability requirements.

Within the framework of this study, a convenient and effective software interface that performs data exchange using the REST API was developed and implemented. This interface is the result of the integration of key technologies that provide its functionality and performance. To ensure the flexibility and reliability of the server part, Flask, a microframe for Python, was chosen. Flask provides a simple and flexible way to process incoming HTTP requests and generate appropriate HTTP responses. When a client sends a request to a specific URL, Flask uses routing to determine which handler function should be called for this request. Inside the handler function, you can perform the necessary operations, access the database, generate data for the response and return an HTTP response to the client.

The frontend part of the client interface was developed using the Angular framework. It provides tools for creating modern dynamic user interfaces, as well as provides effective interaction with the server and data manipulation. An Angular application is built from a set of components, each one representing a specific part of the user interface. Components can be nested into each other, forming a hierarchy. Each component includes an HTML template for displaying the user interface, TypeScript code for logic and data structure, as well as CSS styles for appearance.



Fig. 2: Web interface

RabbitMQ is a message broker that provides efficient data exchange between application components. It allows you to send, receive and route messages asynchronously, which makes it a useful tool for building distributed and scalable systems. RabbitMQ guarantees message delivery and various types of exchanges for flexible routing. This message broker is widely used in microservice architecture and event processing systems.

Celery is a system that allows you to perform operations in the background. It allows you to create and send tasks to a queue for asynchronous execution, which is useful for performing long operations such as sending emails, image processing and other computational tasks. Celery integrates with various message brokers and provides flexible configuration for managing tasks in a distributed environment. This tool is often used to create scalable, reliable, and high-performance applications.

The joint use of RabbitMQ and Celery when creating a system for asynchronous task processing helps to improve its performance. Their interaction also contributes to the division of responsibilities between the components of the system. Flask can focus on HTTP request processing while task processing is passed to RabbitMQ and Celery, providing a modular and flexible application architecture.



Fig. 3: RabbitMQ operation diagram

The integration of all these technologies made it possible to create a powerful and flexible interface for data exchange, contributing to more efficient analysis and management of information in the EventIndex system.

DATA GENERATION

To test the prototype of the system, sets of generated intermediate Event Index data are created. The format of these sets is JSON, and it does not depend on the format in which the data from the detector will be stored. For each event, identifiers are generated (run\_number and event\_number), a random of\_result, as well as fuji\_raw and psd\_raw. In the future, pseudo-data for AOD files is created based on these sets for the same events.

This data is then written to tables in the database: event records are stored in the "events" table, and information about datasets is stored in the "datasets" table. The dataset ID serves as a foreign key for the event table. Pseudo-data generation and writing to tables is carried out using a Python script.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| run\_number | event\_number | olf\_result | dsid\_raw | fuid\_raw |
| 280308061 | 12 | 21810 | 1 | dfcdd203-a473-4c0c-909d-a35252512595 |
| 280308061 | 30 | 17705 | 1 | dfcdd203-a473-4c0c-909d-a35252512595 |
| 280308061 | 43 | 19125 | 1 | dfcdd203-a473-4c0c-909d-a35252512595 |

Fig. 4: the "events" table

|  |  |
| --- | --- |
| dsid | dsn |
| 1 | data28\_pp\_9p4GeV.280308061.RAW.t0001 |
| 2 | data28\_pp\_9p4GeV.280308123.RAW.t0001 |
| 3 | data28\_pp\_9p4GeV.280308042.RAW.t0001 |

Fig. 5: the "datasets" table

With large amounts of data, it may be difficult to process, transmit and analyze information in real time. In the next section, we will consider methods that can be used to effectively address issues related to ensuring high data processing speed.

OPTIMIZATION OF DATA RECORDING SPEED

The procedure used in the first version of the interface for recording each event with a separate PUT instruction showed insufficient speed when testing even on relatively small data arrays. A system capable of coping with a flow of tens of thousands of events per second is needed. Various optimization methods have been investigated to speed up the process of writing data to tables.

To increase the speed of working with PostgreSQL, the "Bulk loading" method was used, in which data is loaded in large blocks. There are many approaches that facilitate large-scale import and provide scalability:

1. Replacing the INSERT operation with COPY. Avoids the overhead of organizing multiple write operations and optimizes the creation of indexes.

2. Optimization of control points and the volume of the write-ahead log (WAL).

3. Disabling triggers.

4. Unregistered tables.

5. Optimization of the column order. Placing fixed-length columns in front

The use of these methods should optimize the bulk upload process and ensure more efficient work with bulk data in PostgreSQL. At the same time, it is necessary to take into account the possible impact of optimization on data integrity and the possible need for recycling.

Along with using the "common" psycopg2 driver for interaction with PostgreSQL, several alternative modules were investigated in order to determine the best option for subsequent optimization. Currently, the following modules have been tested: asyncpg, aiopg, pg8000, PyGreSQL, py-postgresql, SQLAlchemy.

In the original version of the code, the generated events were written to the database one by one through INSERT queries. Reading and writing a small number of events was performed quickly enough for all modules (For example: 10 events, on average, are recorded in 1.647 seconds). With an increase in the amount of data, the recording speed slowed down significantly (For example, 1,000 events, on average, are recorded for 1 minute 55.789 seconds). This data recording speed was not suitable for our task, given that it is expected to receive about 30 billion events per year.

At the first stage, the bulk loading method was used to optimize the program, in which events were recorded into the database in a single block for one INSERT request. The results are shown in Figure 6. The graph shows that this method of recording data is not suitable for asynchronous modules, since it takes them more than two hours to work with 100 thousand events. The fastest module for both small and large amounts of data is psycopg2.



Fig. 6: Time of data set recording using various modules via INSERT query.

At the second stage, to optimize the speed of data writing, a COPY request was used instead of an INSERT request. Testing of the modules showed that py-postgresql and aiopg are not suitable for working with the bulk upload method via a COPY request. The test results of the remaining five models are shown in Figure 7.



Fig. 7: The recording time of data sets using various modules via a COPY request.

Using COPY accelerated the writing of data to the table by about 2 times. It follows from the graph that when working with a small number of events (from 10 to 100,000), using the PyGreSQL module is effective, but for a large number of events (from 1,000,000), the asynchronous asyncpg module should be used. The recording time of 1 and 10 million events is presented in more detail in Figures 8 and 9.



Figure 8: The time of recording 1,000,000 events using various modules via a COPY request.



Fig.9: Recording time of 10,000,000 events using various modules via a COPY request.

At the moment, work is underway to optimize the data write speed by changing PostgreSQL parameters. It was possible to slightly increase the write speed (for 1 million by an average of 3-5 seconds and for 10 million by about 1-1.5 minutes) by changing the "maintenance\_work\_mem", which sets the maximum amount of memory for database maintenance operations. By default, the parameter value was 65.536 MV. Testing was performed for 1 and 10 million events with "maintenance\_work\_mem" equal to 100.536 MV. The results are presented in Figures 10 and 11.



Figure 10: Comparative analysis of data recording speed before and after changing the "maintenance\_work\_mem" parameter for 1 million events.



Figure 11: Comparative analysis of data recording speed before and after changing the "maintenance\_work\_mem" parameter for 10 million events.

In addition, a change in the "synchronous\_commit" parameter led to an increase in the speed of reading and writing data, which determines after the completion of which level of processing the WAL server will report the successful completion of the operation, as well as whether the server should wait for the corresponding WAL records to be processed on the slave server(s) when committing the transaction. By default, the value is set to "on", at which the commit is completed only after a response is received from the current synchronous slave servers confirming that they have received a record of the transaction commit and transferred it to a reliable repository. If you set the value to "off", performance will increase, but only by a few milliseconds, but this may lead to a decrease in data security in the event of a failure. Therefore, it is better to leave the "synchronous\_commit" parameter unchanged. The same result was obtained when changing the "wal\_level", which is responsible for how much information is written to the WAL log, so it also remained unchanged.

Parameters such as «max\_wal\_size», «checkpoint\_timeout», «effective\_io\_concurrency» and «max\_wal\_senders» were also left with their original values, since their decrease or increase led to a decrease in the speed of program execution.

Currently, the effect of block size on download speed is being investigated. The system tries to load all the data at once, but at high load it adapts using a cluster approach for efficient data processing. Additionally, the possibility of parallel data loading is being considered to improve performance. All these methods will optimize data speeds and provide more efficient information processing.

CONCLUSION

In the course of further development of the EventIndex project, the following tasks are expected to be performed:

1. Development of user authorization and authentication mechanisms using single sign-on technology, as well as implementation of group access policies.

2. API development (REST. Python, C++) for use in processing and analysis programs

3. Optimization of processing user requests, with the output of results in synchronous or asynchronous form, depending on the amount of data requested.

4. Development of mechanisms for transmitting "EventIndex" data obtained by indexing files located on remote nodes of the computer network.

5. Develop a supervisor - software for managing the collection and import of data into "EventIndex".

6. Development of a monitoring system for the "EventIndex" component, with the data represintation in graphical form using common platforms (Grafana, etc).

The implementation of these tasks will be carried out in cooperation with the development of other IS installations and depending on their readiness

BIBLIOGRAPHY

* [The ATLAS EventIndex: A BigData Catalogue for All ATLAS Experiment Events](https://arxiv.org/ftp/arxiv/papers/2211/2211.08293.pdf), Computing and Software for Big Science (2023) 7:2
* [Information systems for data taking and data processing, SPD collaboration meeting, 2023](https://indico.jinr.ru/event/3575/contributions/20669/attachments/15249/25742/2023-04-26.Information%20systems%20for%20data%20taking%20and%20data%20processing.pdf)
* [TechnicalDesignReport\_SPD2023](http://spd.jinr.ru/wp-content/uploads/2023/03/TechnicalDesignReport_SPD2023.pdf)
* [PostgreSQL: Documentation: 15: 30.5. WAL Configuration](https://www.postgresql.org/docs/current/wal-configuration.html)
* [PostgreSQL: Documentation: 15: Chapter 20. Server Configuration](https://www.postgresql.org/docs/current/runtime-config.html)
* [Seven practical tips for mass uploading data to PostgreSQL / Habr](https://habr.com/ru/articles/519788/)

ACKNOWLEDGMENTS

The author expresses gratitude to Fedor Prokoshin, Inga Tvauri and Richard Gurtsiev for providing comprehensive guidance and moral support in the process of work, as well as for technical support and valuable recommendations. I am also grateful to the organizers of START for the opportunity to complete the program.