Journal of Big Data

RESEARCH Open Access

Big Data architecture for intelligent maintenance: a focus on query processing and machine learning algorithms

Claude Lehmann[1*], Lilach Goren Huber[1], Thomas Horisberger[2], Georg Scheiba[2], Ana Claudia Sima[1] and Kurt Stockinger[1]

*Correspondence: claude.lehmann@zhaw.ch
[1] Zurich University of Applied Sciences, Obere Kirchgasse 2, 8400 Winterthur, Switzerland
Full list of author information is available at the end of the article

Abstract
Exploiting available condition monitoring data of industrial machines for intelligent maintenance purposes has been attracting attention in various application fields. Machine learning algorithms for fault detection, diagnosis and prognosis are popular and easily accessible. However, our experience in working at the intersection of academia and industry showed that the major challenges of building an end-to-end system in a real-world industrial setting go beyond the design of machine learning algorithms. One of the major challenges is the design of an end-to-end data management solution that is able to efficiently store and process large amounts of heterogeneous data streams resulting from a variety of physical machines. In this paper we present the design of an end-to-end Big Data architecture that enables intelligent maintenance in a real-world industrial setting. In particular, we will discuss various physical design choices for optimizing high-dimensional queries, such as partitioning and Z-ordering, that serve as the basis for health analytics. Finally, we describe a concrete fault detection use case with two different health monitoring algorithms based on machine learning and classical statistics and discuss their advantages and disadvantages. The paper covers some of the most important aspects of the practical implementation of such an end-to-end solution and demonstrates the challenges and their mitigation for the specific application of laser cutting machines.

Keywords: Prognostics and health management, Intelligent maintenance, Big data architecture, Heterogeneous data integration, Stream processing, Query processing, Machine learning

Introduction

Producers and owners of industrial equipment have been showing a growing interest in implementing intelligent maintenance solutions. This is due to a combination of reasons, such as the constantly growing availability of commercial solutions for data acquisition, transmission and storage, as well as the recent development of computationally efficient machine learning algorithms. The need for end-to-end solutions for data-driven decision support systems enabling intelligent maintenance spreads over many different fields of industry and infrastructure assets. Such end-to-end solutions are expected to integrate all of the necessary building blocks, starting from data acquisition, onto streaming and data warehousing, through data processing, analytics and visualization of condition monitoring data up to smart maintenance decision support.

The field of decision support for intelligent maintenance ranges from simple condition monitoring, through fault detection, up to fault diagnosis and prognosis. The latter, sometimes termed "Predictive Maintenance" is nowadays only rarely possible due to a very high variability of operating conditions compared to the low availability of data which is representative of all of these conditions. However, fault detection and diagnosis algorithms strive towards prediction and are being implemented in various industrial systems. In most cases, a "plug-and-play" generic solution is inadequate to detect faults or degradation in complex systems. This is due to the uniqueness of the physical systems and their potential critical failure modes. Moreover, these systems produce large amounts of heterogeneous data streams that pose considerable challenges in efficiently storing and analyzing data at scale.

Therefore, the development of an end-to-end architecture that enables highly efficient processing of system-specific data analysis as well as fault detection algorithms, requires the collaboration of engineering domain experts, big data experts, and data analytics experts. The major challenge is how to design a big data architecture for intelligent maintenance that enables efficient query processing as well as data analytics to monitor the health of the systems, identify potential system faults early on, and in the future allow for the prediction of remaining useful life of critical components.

In this paper we present an example for the design of such an end-to-end intelligent maintenance system, which has been developed for industrial laser cutting machines. The big data architecture enables efficient data storage, data streaming, query processing and data analytics for early fault detection.

The paper makes the following contributions:

- We present a detailed use case of an end-to-end IoT solution for predictive maintenance. Previous discussion of the topic in the literature have always been limited to one of the aspects: big data architecture, storage design, analytic pipeline or machine learning algorithms. In this paper we address all aspects together under one roof. In particular, in each of these aspects we outline important pitfalls that are common to practical applications. The solutions that we suggest here can be used as reference for similar intelligent maintenance IoT systems in diverse application fields.
- We present a scalable IoT data analytics pipeline based on a *Big Data architecture* which integrates heterogeneous data streams across an entire fleet of laser cutting machines. Our system enables near real-time and/or discrete interval-based machine health monitoring.
- We perform an *in-depth evaluation of different physical storage design choices* such as partitioning and Z-ordering to enable efficient, multi-dimensional query processing.
- We discuss the main challenges and lessons learned from implementing the IoT data analytics pipeline within an industrial setting using state-of-the-art Big Data technology, such as Azure Databricks, Spark Structured Streaming and Delta Lakes.
- Finally, we demonstrate the possibility of early fault detection in the optical system of laser cutting machines. In particular, we applied both *statistical methods and con-*

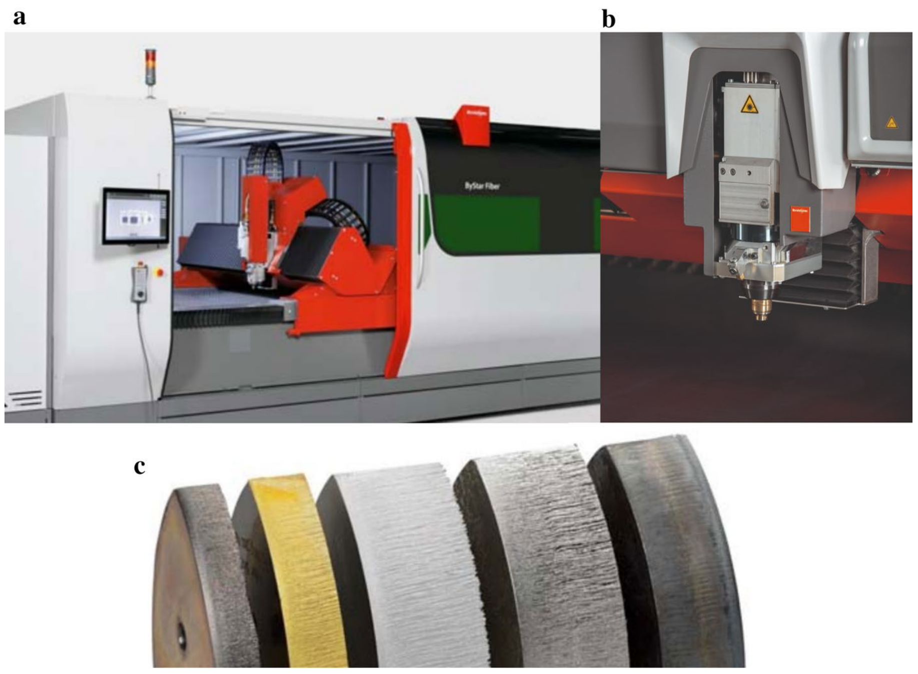


Fig. 1 Intelligent maintenance: **a** A laser cutting machine, **b** the cutting process and **c** examples of materials and thickness

volutional neural networks (CNNs). We claim that although CNNs do not require a first step of feature engineering, standard statistical methods are preferred in this case, due to their higher explainability as well as robustness towards variable operating conditions.

Background

Bystronic is a provider of sheet metal processing solutions. These solutions are targeting the automation of the full customer process, i.e. managing the material and data flow, as well as insuring high availability of the cutting and bending process chains. Figure 1 gives an example of a typical laser cutting machine, the cutting process and material samples. Bystronic collects and analyzes sensor and operation data produced by their fleet of machines for condition monitoring, guided troubleshooting and intelligent maintenance both internally and for their clients.

In order to provide a robust and scalable IoT pipeline for data collection, aggregation and analysis, we designed and implemented a scalable big data architecture shown in Fig. 2. The major components of our architecture are as follows:

- Machine data collection
- Data persistence
- Data processing and analysis
- Data visualization and automated alerting

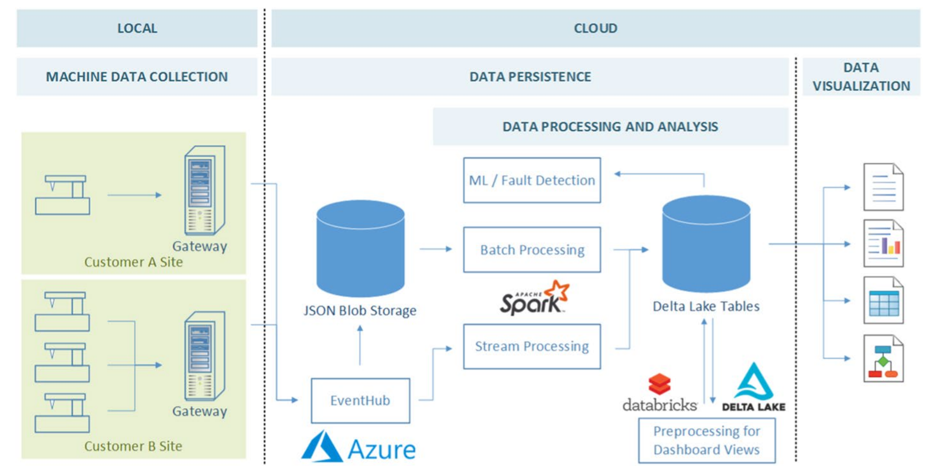


Fig. 2 Schematic view of the system architecture and the process flows

In the following, we explain these components in more detail and provide more background information on the case study of intelligent maintenance.

Component 1: Machine data collection. This component (referred to as gateway) is responsible for retrieving data from the laser cutting machines. Each gateway collects large volumes of heterogeneous data stemming from multiple types of sensors. Some examples are information on temperatures of various parts of the laser cutting machine, spatial coordinates of the cutting head against time, the vibrations of specific components, cutting process feedback, hours of machine use.

Data rates are as high as 1000 events per second. In order to efficiently process this high data volume, special programmable hardware based on Field Programmable Gate Arrays (FPGAs) is used to filter the data according to certain criteria. These FPGAs are attached to the laser cutting machines. The remaining data is then forwarded to the Bystronic domain in the Microsoft cloud in order to enable big data fleet statistics of potentially high data volumes and heterogeneity. The latter is caused by machines being operated under various conditions and in all phases of their service life-cycle.

Component 2: Data persistence. Once the machine data is received on the cloud platform via the EventHub, it is saved as a blob in JSON format - to create the immutable data lake. The EventHub can be configured to persistently store data for a specified time window. For each customer, a separate blob is generated and maintained.

Component 3: Data processing and analysis. Depending on the use case, different processing steps are performed to analyze the machine data, either on the fly in the data stream or in batch processing intervals. For example, Remaining Useful Life (RUL) indicators are calculated in batches at regular intervals to estimate the lifetime of particular parts of the machines (see subcomponent "Batch Processing" in the center of Fig. 2). Furthermore, computing health indicators may require the deployment of statistical methods, signal processing or machine learning algorithms on a single machine or on a fleet data.

As an alternative to batch processing, more critical machine health monitoring may be performed in (near) real-time using Spark Streaming (see sub-component "Stream Processing"). Note that in our use case, stream processing is not time critical.

The stream processing engine is also used to continuously update the so-called "Delta Tables" of Databrick's (mutable) Delta lake in order to enable efficient multi-dimensional query processing and analytics. These Delta Tables are essentially a column store with support for data partitioning and clustering. In condition monitoring tasks, the Delta Tables form the foundation for all dashboard views, where entire time-series data needs to be visualized. The Delta Tables are regularly optimized for peak performance. Recurring jobs are set up to prepare and aggregate data for the statistics dashboard views.

Component 4: Data visualization and automated alerting. In a final step, the condition monitoring and health assessment results are visualized in dashboards and email alert messages sent to the asset stakeholders. Since these kinds of dashboards are used by the field service support teams to define service tasks, the data always needs to be up to date and accurate, which makes stream processing in some use-cases a key requirement. Nevertheless, it is also true, that most condition monitoring use-cases require only daily batch processing of collected data as the machine condition typically deteriorates over longer time scales.

An important design goal of the architecture described above is to keep the number of components as small as possible such that they can easily be maintained by a small operations team. There are certainly different design alternatives. For instance, Apache Kafka[1] could be used to enhance the streaming component and MongoDB[2] could be used for data storage – as we have done in previous industrial-strength use cases [1]. Another alternative would be to use Apache Nifi[3] for distributed data processing. However, we decided against these tools to keep the technology stack simple.

Related work

The use of big data technologies for machine learning-based predictive maintenance applications has been drawing attention over the last couple of years. Several approaches use big data technology to enable processing of large data sets often in combination with machine learning to analyze the data [2–4]. However, our work differs from these approaches, suggesting a more holistic approach that includes tackling real-world challenges. One example for such a challenge are problems due to data scheme changes. Another one, which is particularly relevant for health monitoring, is the processing of multi-dimensional queries.

In this section we focus the literature survey on the central topics of the experimental part below: multi-dimensional indexes for speeding up query processing and machine learning algorithms for intelligent maintenance.

[1] https://kafka.apache.org/.

[2] https://www.mongodb.com/.

[3] https://nifi.apache.org/.

Speeding up query processing with multi-dimensional indexes

With growing data sizes, the idea to narrow down the amount of data that needs to be accessed during query processing is key. This idea of data skipping is achieved in traditional database systems by traversing a B-tree index [5, 6] and selecting only the relevant rows for a given query in one dimension. However, this problem grows more complex for a multi-dimensional use case. Other approaches are necessary to efficiently apply indexing to accelerate multi-dimensional queries.

Examples of multi-dimensional indexes include tree-based data structures (e.g. quad-trees [7], k-d-trees [8], R-trees [9]), space filling curves (e.g. Hilbert [10] or Morton/Z-order curve [11]) and bitmap indexes [12, 13]. Through the success of deep neural networks, learning indexes have also become an option [14].

MongoDB and Neo4j use B-trees for multi-dimensional indexing [15, 16]. Apache Hive added support for bitmap indexes in version 0.8 and dropped them with release 3.0 again [17]. Furthermore, PostgreSQL uses bitmap indexes since version 8.1 to combine multiple existing indexes [18]. Other architectures have successfully been designed around the use of bitmap indexes [19]. Amazon Redshift uses Z-ordering to arrange points [20], Amazon DynamoDB provides Z-order indexes to arrange a multi-dimensional space in one dimension [21]. Moreover, Spark-SQL provides Z-order clustering to physically arrange similar data in the same set of files [22].

A downside of the physical data layout, such as the Z-order clustering used by Spark-SQL, is that it does not allow multiple Z-orderings. This means that unlike with classical indexes, we are unable to create a new data layout for every important query. Instead, a single layout is needed that works best for a general case and still performs adequately in a worst case scenario.

Machine learning for intelligent maintenance

The application of machine learning and deep learning algorithms has drawn a lot of attention in the field of prognostics and health management. There are several review articles that list related work in this subject, see for example [23]. In particular, using CNNs for fault classification is common, especially for high frequency vibration data, e.g. [24, 25]. In this case, some papers use transformation schemes in order to obtain two dimensional images out of the time series data and then apply CNN models that have proven to yield good classification performance on images. Other papers use the raw time series data with 1D or 2D filters, such as [26, 27]. The application of CNNs for the detection of faults in the optical system has not been demonstrated to the best of our knowledge.

There are several libraries that enable the training and inference of deep neural network models with Spark including contributions of organizations like Yahoo (TensorFlowOnSpark[4]), CERN (Distributed Keras[5]) and Intel (BigDL[6]). Fortunately, also Databricks created an open-source extension for the Spark framework called Deep

[4] https://github.com/yahoo/TensorFlowOnSpark.

[5] https://github.com/cerndb/dist-keras.

[6] https://github.com/intel-analytics/BigDL.

Learning Pipelines[7] that seamlessly integrates Keras and TensorFlow. The model training can thus benefit from the tools provided in the Databricks environment by using all existing fleet data readily available in Delta tables.

Methods: system design

In this section we describe the major design choices of our Big Data architecture. In particular, we focus on efficient batch and stream processing, checkpointing for mitigating stream processing failures and query processing. All of these methods have been implemented in a productive system and have thus been verified in a real world scenario.

Monitoring system with stream processing

The stream processing pipeline is designed primarily to provide an opportunity for fast online diagnosis following a machine failure, by collecting the most relevant machine health parameters in real time. The stream processor also serves to convert incoming JSON messages into structured Parquet files (stored in Delta tables—see Fig. 2). However, monitoring parameters across the fleet of machines is a challenging task, not only due to possible scalability issues, but also due to the heterogeneity of the data arriving from the different machines.

We can distinguish between two main types of heterogeneity:

1. Data type heterogeneity—each machine provides data from multiple sensor types. At a gateway, all the different data types are periodically collected and sent via the same stream to the cloud, in JSON format. However, the stream receiver needs to correctly split these messages back by datatype and store each to the appropriate Delta table, given that each type of sensor data is used in a different analysis pipeline.
2. Software version heterogeneity—i.e., schema heterogeneity for *the same* data type *across* software versions. Bystronic provides regular software updates for its machines. Following an update, the structure of the machine data collected may change. For example, an offline analysis of sensor data may show that it is useful to collect and monitor new parameters in order to improve the accuracy of the Remaining Useful Life estimation for a given machine component. As a consequence, the original schema for this data type will be changed, through the addition of the new parameters. This schema change will be applied as part of a software update. However, it is then up to each customer to apply the update either sooner or later. In practice, this means that at every point in time a variety of different software versions will co-exist across the fleet of machines. The stream processing task needs therefore to be able to handle this heterogeneity and correctly parse each data type. Moreover, the machine will not send a specific message when it has been upgraded—it is therefore the responsibility of the downstream tasks to monitor for this change and adapt upon detecting an upgrade.

In order to address the above possible heterogeneity issues, we implement the following general mechanism for processing incoming JSON messages:

[7] https://docs.databricks.com/applications/deep-learning/single-node-training/deep-learning-pipelines.html.

1 Before starting the stream processing job, Delta tables are created for each machine data type collected at the gateway. For each data type, the corresponding schema is stored locally and is used for parsing the received JSON messages into structured dataframes.
2 A stream processing task is launched every trigger interval. In our current setup, we have set the micro-batch trigger interval to 20 seconds. More details on micro-batch trigger intervals are provided in the Spark user guide[8].
3 For each micro-batch, received messages are filtered by data type. This filtering is done on the text-level, i.e. directly on data received on the stream, since different message formats are collocated on the same stream. Filtering by data type will therefore facilitate correctly parsing each JSON message according to the known schema.
4 Parse each JSON message into a structured dataframe, by using the known schema and write the resulting dataframe into the corresponding Delta table.
5 To improve the performance of batch queries executed on the Delta tables, a periodic job is scheduled to optimize the Delta tables. This optimization will also result in compacting the file structure, since writing streaming data into Delta tables can result in a large number of small files being created.

Finally, a batch job checks periodically new data points to verify that the known schema still corresponds to the real schema of the incoming streaming data. This check is to ensure that updates to the input schema do not result in data loss, since any fields not part of the known schema will simply be ignored (dropped). Therefore, in the Delta tables, the inferred schema is also stored along with the actual data. Upon detecting a change, the known schema for the data type is updated in the stream processor, and a recovery mechanism is triggered in order to re-process the data points since the schema changed.

Checkpointing: monitoring the monitoring system

Since the streaming task is responsible for providing up-to-date machine information, it is important for this service to have minimal downtime. However, there are multiple possible failure scenarios for the stream processing pipeline. Here, we briefly discuss a few examples of such failures and the solutions implemented to overcome them in practice. In general, there are two required components for handling failures: monitoring the stream processing pipeline and checkpointing the last successfully processed timestamp per machine.

There are two main types of possible failures in the stream processing pipeline:

1 An error in the stream processing task:

- This can happen due to an unexpected runtime error, for example, caused by memory issues within the streaming task.
- We can consider this a soft failure, since the task can be restarted and the data will be subsequently re-processed through Spark's own checkpointing mechanism

[8] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#triggers.

(assuming the task is restarted within the buffering window of the stream). For more details on checkpointing in Spark, see the Spark user guide[9].

- For this type of error, it is sufficient to monitor the stream in a separate task, by periodically investigating its status. Whenever the status is detected as "Stopped", the streaming task should be restarted. The last successfully processed timestamp will be available through the "lastProgress" parameter of the stream[10].

2 A cluster failure

- This is a more significant failure, since all intermediate states will be lost, which implies that we may no longer be able to rely on Spark's checkpointing mechanism. Instead, in this case we need to internally store the last successfully processed timestamp for each data type in order to overcome gaps in the data. More precisely, upon restarting the cluster, a batch job will process, from the immutable JSON data lake, all the data points between the last saved timestamp and the time of the restart and store them in the appropriate Delta tables. This will ensure that all the data is also available in structured format, despite the cluster downtime.

Finally, although not strictly a failure, a software update that results in input schema changes will also require restarting the stream (and therefore will result in some downtime). As machines do not have any means of signalling a software update, we need to monitor for schema changes downstream, when processing data. Importantly, the stream processing receiver expects the input to match the known schema, which means that any new parameters added will be discarded. For this reason, the schema monitoring will periodically analyze new data points, infer their schema and compare it against the known one. Upon detecting a change, the stream processor will update its information and restart. To minimize the impact of this downtime on processing data from other machines, the stream can be partitioned either by machine or by data type (since all information from one type, across the fleet of machines, will be written to the same delta table).

Query processing and optimization

An important data preprocessing step amounts to an efficient extraction of relevant data subsets for a particular analytical task. This extraction is done through data queries. For example, if training a fault detection algorithm requires data from one specific machine "M4" over a limited period of time of the year 2019, we need to query the entire database for the point value Machine="M4" and for the range value *time* >= 1.1.2019 and *time* < 1.1.2020. This is an example of a *two-dimensional* query in which we search both the machine dimension and the time dimension. The machine dimension typically has a small set of possible values it can assume (order of hundreds or thousands of machines)—it is therefore a *low cardinality attribute*. In contrast, the time dimension can typically assume a very large set of values, depending on the data resolution. The time attribute is thus a *high cardinality attribute*. Moreover, in this example the query

[9] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#recovering-from-failures-with-checkpointing.

[10] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#managing-streaming-queries.

type is mixed: one of the dimensions (machine) is a *point query* and the other one (time) is a *range query*.

As a generalization of the above example, we can characterize queries with the help of three main input dimensions: (1) *data density*: low-attribute cardinality vs. high-attribute cardinality (2) *query type:* point query vs. range query (3) *query dimensionality*: one-dimensional query vs. multi-dimensional query.

The data storage layout crucially influences the response time of queries. Each of the three input dimensions may have an important effect on the response time, depending on the queries that are frequently executed. We should therefore aim at finding the *optimal storage technique* as well as the optimal parameters for this storage technique such that the response times of queries are minimized. The problem can thus be formulated as an optimization problem across the three *input dimensions* listed above.

Input dimensions

We will now discuss these three input dimensions in more detail, while analyzing the impact of two storage optimization techniques provided by Apache Spark, namely *data partitioning* and *Z-ordering* [11].

Data density: The cardinality of an attribute refers to the number of unique values. From the point of view of query optimization, we distinguish between attributes with a *low cardinality*, e.g. number of days in a week, and attributes with *high cardinality*, e.g. temperature values of a machine. Depending on the attribute cardinality, different storage optimization techniques need to be chosen.

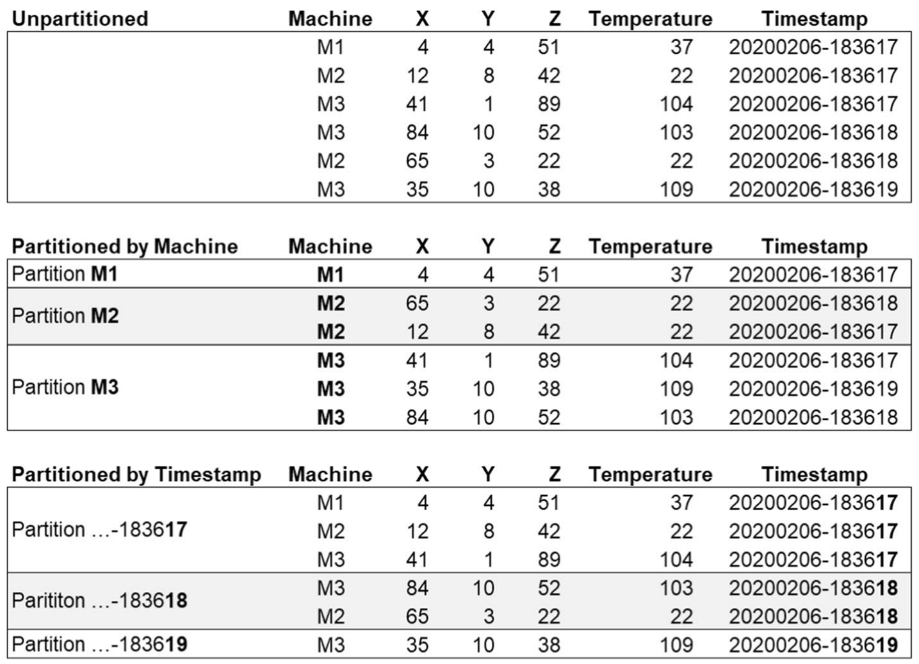
Query type: We can distinguish between two different types of queries, namely *point queries* and *range queries*.

Point queries (PQ) are of the form $a = v$ where a refers to an attribute and v to a value. For instance, find all machines where a failure occurred on weekday $= 5$. Range queries (RQ) are of the form $a > v$ (greater than) or $a < v$ (less than). For instance, find all machines with a temperature > 100.

Query dimensionality: We can distinguish between *one-dimensional* and *multi-dimensional queries* depending on how many different attributes are contained in the query. For instance, the query of the form $a = v$ is a one-dimensional query, while the query of the form $a_1 = v_1$ AND $a_2 = v_2$ is a two-dimensional query.

In the real-world, queries are often complex. Hence we *combine* query type and query dimensionality:

- *One-dimensional point queries (1D-PQ)* are of the form $a = v$ where a refers to an attribute and v to a value.
- *One-dimensional range queries (1D-RQ)* are of the form $a > v$ (greater than) or $a < v$ (less than).
- *Multi-dimensional point queries (nD-PQ, i.e. 2D-PQ)* combine multiple one-dimensional point queries with the boolean operator *AND*. They are of the form $a_1 = v_1$ *AND* $a_2 = v_2$ *AND ... AND* $a_n = v_n$. For instance, find all machines where a failure occurred on weekday = 5 AND location = Zurich.
- *Multi-dimensional range queries (nD-RQ, i.e. 2D-RQ)* - similar to multi-dimensional point queries - are a combination of one-dimensional range queries, concatenated



Unpartitioned	Machine	X	Y	Z	Temperature	Timestamp
	M1	4	4	51	37	20200206-183617
	M2	12	8	42	22	20200206-183617
	M3	41	1	89	104	20200206-183617
	M3	84	10	52	103	20200206-183618
	M2	65	3	22	22	20200206-183618
	M3	35	10	38	109	20200206-183619

Partitioned by Machine	Machine	X	Y	Z	Temperature	Timestamp
Partition **M1**	**M1**	4	4	51	37	20200206-183617
Partition **M2**	**M2**	65	3	22	22	20200206-183618
	M2	12	8	42	22	20200206-183617
Partition **M3**	**M3**	41	1	89	104	20200206-183617
	M3	35	10	38	109	20200206-183619
	M3	84	10	52	103	20200206-183618

Partitioned by Timestamp	Machine	X	Y	Z	Temperature	Timestamp
Partition …-1836**17**	M1	4	4	51	37	2020206-1836**17**
	M2	12	8	42	22	20200206-1836**17**
	M3	41	1	89	104	20200206-1836**17**
Parititon …-1836**18**	M3	84	10	52	103	20200206-1836**18**
	M2	65	3	22	22	20200206-1836**18**
Partition …-1836**19**	M3	35	10	38	109	20200206-1836**19**

Fig. 3 Partitioning: Top part shows a dataset with measures of several machines. Middle and lower parts show how the data is rearranged when partitioned by `Machine` and `Timestamp`, respectively

with the boolean operator *AND*. They are of the form $a_1 > v_1$ *AND* $a_2 > v_2$ *AND* ... *AND* $a_n > v_n$. For instance, find all machines with a temperature > 100 AND speed < 20.

Data storage optimization

Given the three aforementioned input dimensions, the goal is to find the optimal data storage technique to minimize query execution time. Modern database management systems and data warehousing solutions provide various techniques for optimizing access to data, such as data partitioning [28] or indexes [29]. However, Apache Spark and in particular Databricks do not support indexes. In order to optimize queries, Spark provides data partitioning and Z-order clustering. [22].

- *Partitioning* is used to split files into more manageable sizes such that a smaller subset of the data needs to be accessed during query processing. Let us assume a dataset with measurements for various machines. These measurements contain the machine name, the three coordinates of the cutting head of the machine (x, y and z), the temperature values of a particular machine part, and a timestamp of when the measurement was collected (see top part of Fig. 3). The middle and bottom part of the Figure show two data partitioning strategies, namely, partitioning by machine and partition by timestamp.

  Partitioning by machine is a good strategy for queries that look for specific machine names. On the other hand, partitioning by timestamp is the better strategy for

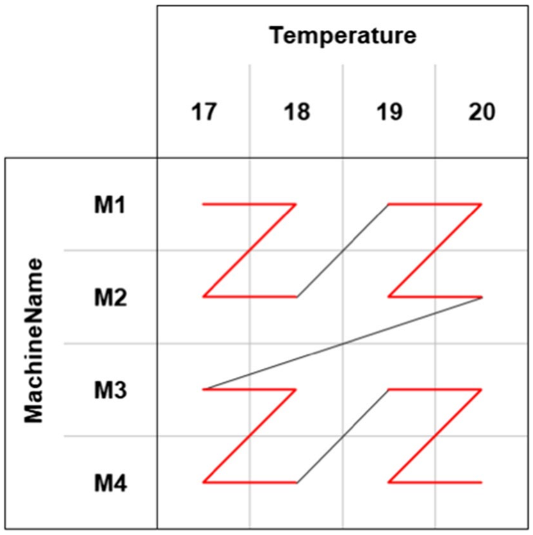


Fig. 4 Z-ordering: Example of how the data shown in Fig. 3 is rearranged based on a two-dimensional Z-ordering of the attributes `Machine` and `Temperature`

queries looking for timestamps. However, what if we have a query that looks for machines AND timestamps? In this case, one might need a partition strategy that *combines* the partitions for machines and timestamps. In general, given that our system can receive arbitrary queries with any possible combination of attributes and dimensionality, the complexity of choosing all combinations of partitions is $O(n!)$, where n is the number of attributes in a table[11]. Storing such a high number of partitions is impossible in practice. Hence, one needs to choose the partitions based on the most commonly used queries. This could be achieved only by correctly combining system specific domain knowledge, and allowing for an iterative development process which can adapt the partitions to the most frequently deployed data analytics.

- *Z-ordering/Multi-dimensional clustering* The basic idea of Z-ordering [11] is to cluster those attributes together that are potentially also queried together. In other words, Z-ordering maps data from a multi-dimensional space down to a one-dimensional space. Moreover, Z-ordering uses a *combined partitioning strategy* across multiple attributes. In our example (see Fig. 4), the values of some of the machines are collocated with the values of some of the temperature measurements. The data can now be traversed following the shape of the letter "Z".
  Let us assume that we are interested in the query where Machine = M2 AND temperature < 19. In this case, we only need to traverse the top left Z. However, if we are interested in all machines with temperature < 19, we basically need to traverse three times as much data, i.e. three Z-shapes. This example shows that depending on the type and the dimensionality of the queries, a different amount of data needs to be

[11] Consider table T1 with attributes a1, a2, and a3. We can choose 3 single-attribute partitions (a1; a2; a3), 3 two-attribute-partitions, ((a1,a2); (a1, a3); (a2,a3)), and 1 three-attribute-partition (a1, a2, a3).

traversed. Thus, the more data that needs to be traversed during query processing, the longer the query response time.

Partitioning and Z-ordering are mutually exclusive operations for any given attribute. In other words, for a given attribute, one can either use partitioning or Z-ordering but not both.

It is important to note that these optimizations are applied on the physical file level and cannot be customized on a per-query basis. This means that whichever configuration is initially chosen, will be applied for *all* queries. As time passes, most users discover new queries that are equally important but were not considered when evaluating optimization strategies. As such, it is important to keep in mind that any of the strategies used should not adversely affect other queries.

Results and discussion

In this section we perform an experimental evaluation of the query performance in relation to Spark's physical storage optimization techniques (see "Analysis of query performance in Big Data architecture" section). Afterwards, we will analyze the accuracy of our machine learning algorithms for intelligent maintenance (see "Statistical and machine learning algorithms for intelligent maintenance" section).

Analysis of query performance in Big Data architecture

Given the three input dimensions *data density*, *query type* and *query dimensionality*, we will now evaluate which is the *optimal storage solution* such that the query response time is minimized. To solve the optimization problem across these three input dimensions, we will address the following research questions:

- What is the effect of partitioning and Z-ordering on one-dimensional queries?
- How are the conclusions affected by higher query dimensionalities?
- How do our findings change when we deal with high cardinality attributes?

In our experiments we study the performance of one-dimensional and multi-dimensional point and range queries. The experimental setup is inspired by previous work on analyzing the query performance of Big Data systems [19]).

Software and hardware setup

All the experiments were conducted on Azure Databricks using the latest stable runtime versions, which at the time were Apache Spark 2.4.4 and Scala 2.11. Three different cluster configurations were used with two, four and eight worker nodes, respectively. All cluster nodes are of the $Standard\_DS13\_v2$ type and have 56 GB of memory available, as well as 8 CPU cores each[12]. In total, the Spark cluster has a maximum memory of 448 GB and 64 CPU cores. Autoscaling was disabled for all clusters to ensure constant performance availability. The cluster mode was set to *high concurrency*. All code was written in Python version 3.

[12] See https://azure.microsoft.com/en-in/pricing/details/databricks for cluster configurations and pricing information.

Table 1 Characteristics of the synthetic dataset used for our benchmarks

N_{rows}	10^{10}
N_{cols}	16
Data size	240 GB

Datasets

The goal of the experiment was to compare the influence of optimization strategies on Databricks Delta tables using both partitioning and Z-order clustering. Since, the acquisition of real machine data only started within this research scope and has not yet reached a sufficiently large volume to analyze query performance at scale, we created a synthetic dataset, where attribute values are sampled from uniform distributions. The synthetic dataset has a size of 240 GB. The key characteristics are shown in Table 1.

Our synthetic dataset follows a uniform distribution. For unpartitioned attributes, we set the attribute cardinalities to 10^3 if not specified otherwise, in order to better control our experiments (similar to [19]).

Partitioned attributes are treated differently since each partition results in a folder being created. With multiple partitioned attributes this leads to an immense amount of folders and subfolders in the order of the product of all partitioned attribute cardinalities. To mitigate this effect, we used one, two and four partitioned attributes with cardinalities of 10^4, 10^2 and 10^1, respectively. This results in a similar amount of folders across the different partition configurations and amends the slowness of file systems at listing millions of files [30].

Experiment results

In this section we explore the impact of partitioning and Z-Ordering through a series of experiments. The queries were executed on clusters with two, four and eight worker nodes with 16, 32 and 64 CPU cores, respectively. We executed each query five times and report the average response time. Moreover, we randomly sampled the query values from the available value space of the respective column.

Experiment 1: A comparison between partitioning and Z-ordering for one-dimensional queries In the first set of experiments we compare different optimization strategies on the *synthetic dataset* with attribute cardinalities of 10^4. In particular, we analyze the performance of one-dimensional point and range queries using the following three different table configurations:

- Non-partitioned: The table is physically stored without optimization and specifically without partitioning.
- Z-ordering: The table is stored using a specific attribute for Z-ordering. The Z-ordered attribute is also accessed by the queries.
- Partitioned: The table is partitioned by a specific attribute. This partitioned attribute is accessed by the queries.

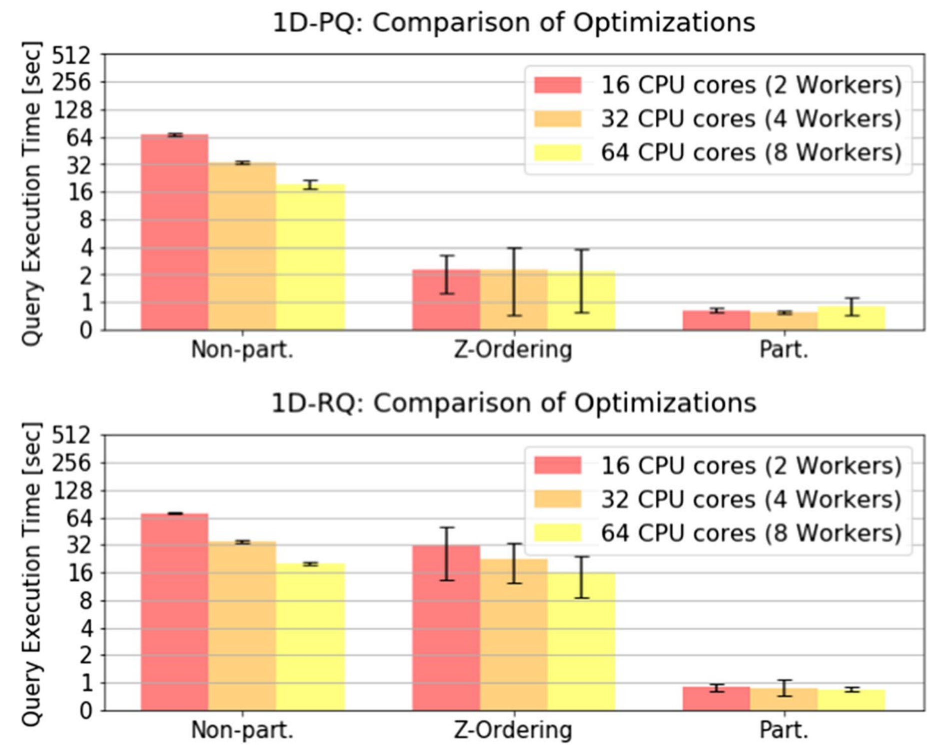


Fig. 5 Response times for 1D point queries (top figure: 1D-PQ) and 1D range queries (bottom figure: 1D-RQ) on the synthetic dataset. The cardinality for all attributes is 10^4. The queries are of the from `WHERE a1 = X`, where *a*1 is a non-partitioned, partitioned or Z-ordered attribute, respectively

Figure 5 shows the query performance using these three configurations. The top part shows the performance of one-dimensional *point queries* (1D-PQ). The bottom part shows one-dimensional *range queries* (1D-RQ). Note that the y-axis is scaled logarithmically to the base 2. We can see that partitioning leads to the lowest response times of less than 1 second both for point queries and for range queries. Z-ordering is slower by a factor of 3 for point, and a factor of 16 to 32 for range queries, depending on the cluster size. The difference for range queries is that on average a larger number of files needs to be analyzed than for point queries. Not utilizing any of the optimizations is slower than partitioning by a factor of 20 to 32. Note that Z-ordering has the highest variance in response times – that is, the strongest sensitivity towards the specific queried values. Moreover, Z-ordered range queries are faster than queries on non-partitioned attributes by a factor of two.

Even though queries against non-partitioned attributes are the slowest, they show almost linear scalability with respect to the cluster size. For range queries against Z-ordered attributes we see similar linear scalability. For queries against partitioned attributes we do not see this positive scalability effect. However, the query response times against partitioned attributes is already below 1 second and hence is hard to further improve due to network latencies and thus increased communication costs between 4 or 8 worker nodes.

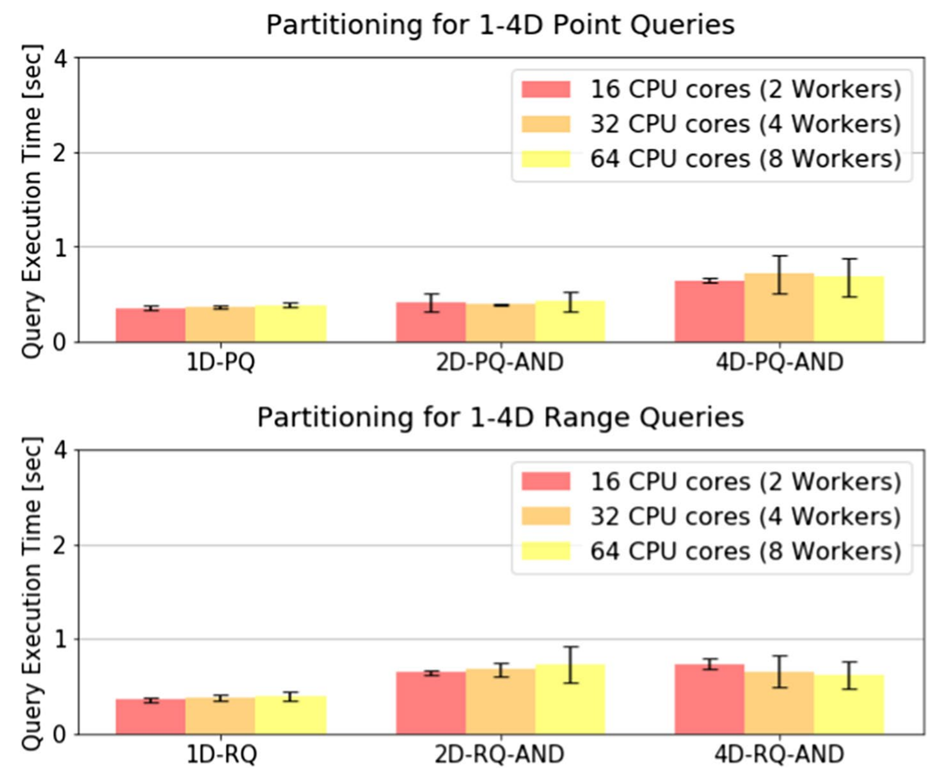


Fig. 6 Response times for one- and multi-dimensional point and range queries against partitioned attributes on the synthetic dataset. The x-axis indicates the dimensionality of the executed query, i.e. `2D-RQ-AND` refers to a 2D range query of the form `WHERE p1 = X AND p2 = Y`

The high query performance of the partitioned table can be explained by the fact that Spark can take advantage of file-level statistics, indicating which attribute ranges are contained in the partition and thus narrowing down the amount of data access during query processing (as they are co-located in the same partition folder). Z-ordering, however, (and especially in the case of range queries) has a disadvantage since rows cannot be selected by only looking at partitions containing the queried attribute ranges. Instead, every file has to be checked individually.

Experiment 2: Multi-dimensional queries against partitioned attributes In our previous experiments we showed that partitioning performs better than Z-ordering for one-dimensional queries. In our next set of experiments we analyze the performance of multi-dimensional conjunctive queries against partitioned attributes.

In particular, we create tables with one, two and four partitioned attributes with attribute cardinalities of 10^4, 10^2 and 10^1, respectively. This means that the cardinality product of all partitioned attributes is 10^4 for all three configurations.

As shown in Fig. 6, for all queries the response times stay below one second, even in the case of range queries. In other words, *increasing the dimensionality of the queries only marginally increases the response times of queries*. In practise, a response time

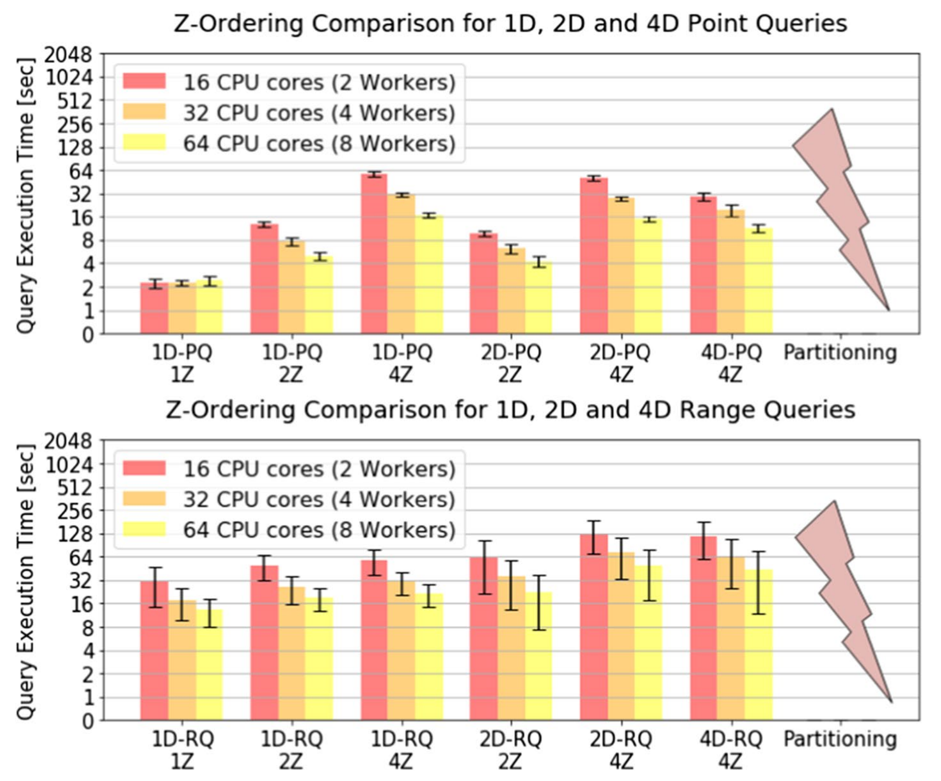


Fig. 7 Response times for Z-ordered attributes with cardinality of 10^6. For instance, 2Z indicates that two attributes were used for Z-ordering. The flash symbol indicates that Spark was unable to create partitions using these attribute cardinalities

below one second is very acceptable. All these queries are very likely limited not by the dataset size but by the computational overhead of Spark or the network latency[13].

Similar to previous experiments with partitioning, the size of the cluster is not influencing the response time in a significant way.

Experiment 3: Querying high cardinality attributes In our previous experiments we used low attribute cardinalities ranging from 10^2 to 10^4. In the next set of experiments we evaluate the performance of queries against attributes with *high cardinalities*. For these types of attributes, data partitioning is not feasible anymore[14] and Z-ordering is the recommended solution.

For our experiments, we perform one- and multi-dimensional point and range queries for one, two and four Z-ordered attributes with cardinalities of 10^6. Figure 7 indicates that for such cardinality regions, Z-ordering shows a strong performance especially when few attributes are Z-ordered. Labels on the x-axis indicate the query type and the table configurations, e.g. “2D-PQ 2Z” refers to a two-dimensional point query against two Z-ordered attributes.

[13] Our experiments showed that even for very small dataset sizes, the minimum query response time of a query is on the order of half a second and cannot be further reduced due to Spark-internal overheads.

[14] At a cardinality of 10^6 Spark was unsuccessful in creating partitioned tables leading to an unmanageable amount of files and folders.

Figure 7 further shows deteriorating performance for queries that do not include all Z-ordered attributes in their `WHERE` clause. Let us first focus on the query response times of point queries. In particular we analyze the effects of these two Z-ordering strategies `1D-PQ 1Z vs. 1D-PQ 4Z`. In the former case, one-dimensional point queries are executed using Z-ordering based on *one attribute*. In the latter case, one-dimensional point queries are executed using Z-ordering based on *four attributes*. We can observe that the query response times on 64 CPU-cores are about 8 times higher for `1D-PQ 4Z` compared to the configuration of `1D-PQ 1Z`. The reason is that Z-ordering aligns rows *maximizing all Z-ordered attributes equally*, such that there is a performance drop when applying Z-ordering to more attributes than those being used in the query. In summary, one-dimensional queries perform better on Z-ordering with one attribute rather than four attributes.

Let us now focus on the response times of *range queries* (see lower part of Fig. 7). We can observe that the response times are significantly higher than for point queries. The reason is that Z-ordering is very sensitive to the *attribute range* covered by the query. Hence, on average, for range queries a larger portion of the entire index has to be scanned as compared to point queries.

Best practices deduced from performance experiments With our experiments we showed that optimizing for attributes that are often queried yields substantial performance benefits. As such we define the following best practices:

- Categorical attributes such as *year*, *month*, *day* or *country* have a *low attribute cardinality*. For these attributes, *partitioning* works well and should be used primarily. As a general rule of thumb, at a cardinality above 10^4 one should start to question, if partitioning is the right choice.
- Continuous attributes such as *temperaturevalues* typically have *high attribute cardinalities* and are best suited for *Z-ordering*.
- In general, partitioning is faster than Z-ordering and should be preferred, if the cardinality is below 10^5. Z-ordering is able to handle high attribute cardinalities where partitioning is not an option anymore.
- Unlike partitioning, the performance of Z-ordering degrades with multiple attributes that are Z-ordered, but not part of a query.
- Our performance experiments with real-world data showed similar results and could confirm our lessons learned.

In conclusion, the optimal strategy is strongly influenced by the nature of the attributes which are more commonly queried, by the characteristics of a typical query workload (point or range queries, aggregations, etc.) and by the decision, where performance is most critical.

Statistical and machine learning algorithms for intelligent maintenance

Use case description

In this section we focus on a specific use-case for data-driven methods for the detection of faulty behavior in the laser cutting machine, that can in turn allow for more intelligent maintenance measures. In particular, we demonstrate the possibility of

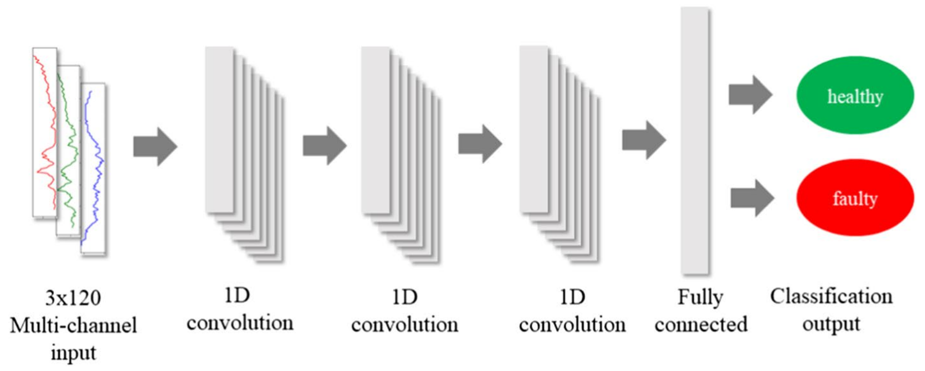


Fig. 8 Convolutional neural network architecture for machine classification

early fault detection in the optical system based on sensor data. We then suggest to train, validate and deploy the derived detection models in the infrastructure presented in the previous sections. This use case is currently based on machine data out of laboratory experiments rather than field data, but is intended as a proof of concept for later implementation with real-time field data collected from a fleet of machines. The main purpose of the presented use case is to detect contamination of optical components that are likely to lead to reduced production quality of the machine. The components can then either be cleaned or exchanged early enough to avoid low quality production outcomes. To this end we developed two alternative fault detection algorithms. The first one uses convolutional neural networks (CNNs) for classification of sensor signals into "healthy" and "faulty". The second algorithm uses conventional statistical analysis and signal processing methods in order to construct direct health indicators for fault detection.

From the perspective of a machine operator, early fault detection must be accurate. Detected faults should correspond to real faults and not mere abnormalities in the data. False positives would result in unnecessary service measures, that would reduce the availability and therefore the productive use of the machine.

Datasets

We collected laboratory data from 14 different machines. 12 of the machines were labelled as healthy and 2 of them as faulty. The implications of a faulty machine are low quality production outcomes under certain operational conditions. Clearly, not all operational conditions have the same impact on the production outcomes. However, the aim of the present work is to detect underlying faulty behavior even under conditions which might be interpreted as benign. The raw data is obtained with a maximal time resolution of 1msec and is then down sampled to a time resolution of 100msec. We extracted input sequences of length 120 (corresponding to 12 seconds) with partial overlap (shifted by 200msec). These parameters were selected to optimally capture the time correlations of the input series and to maximally augment the data. This yields a total amount of training sequences of 38,990 and a total amount of testing sequences of 304,507. The purpose here is to demonstrate that there is no need in very large amounts of training data to

Table 2 CNN classification performance

Machine	TN	FN	TP	FP	Accuracy
1	10963	0	0	0	1
2	13133	0	0	0	1
3	8817	0	0	0	1
4	7502	0	0	0	1
5*	0	2978	90876	0	0.968
6	7690	0	0	0	1
7	22899	0	0	21	0.999
8	28724	0	0	11	0.999
9	52533	0	0	0	1
10	3636	0	0	504	0.878
11*	0	184	23556	0	0.992
12	14933	0	0	0	1
13	5491	0	0	31	0.994
14	9992	0	0	33	0.997

obtain good results on a large and diverse test set, as long as the training data is representative enough.

Method 1: Machine classification based on convolutional neural networks

We construct a CNN for the binary classification of multivariate sequences (time series) into "healthy" and "faulty". The goal is to identify faulty machines, which are defined as machines with more than 90% faulty samples. The CNN receives multivariate time series data as an input and delivers a binary output: healthy or faulty for each input sample.

*Network architecture We used a multivariate input sequence with 3 input variables, chosen based on engineering domain knowledge (see Fig. 8). The input is fed into three consecutive convolutional hidden units. Each unit contains a 1D convolutional layer with 10 filters, a dropout layer classifier and a batch normalization layer. The binary fully connected layer contains 80 units. For the implementation we used Keras of Tensorflow[15].

The choice of a CNN architecture is done in order to exploit time dependent features of the sensor data. Comparing the results to a standard fully connected neural network (a Multilayer Perceptron) indicated a clear superiority of the CNN. A valid alternative would be Recurrent Neural Networks (RNNs). However, they require a more elaborate hyperparameter search and have been shown to perform similarly or even worse than CNNs for sequence modelling [31].

Training For training we used time sequences out of healthy as well as faulty machines. In each experiment we used data from only one of the two faulty machines as well as 4 healthy machines for training ("training machines"). The rest of the machines were used exclusively for testing ("testing machines"). We randomly selected parts of the data of the training machines for training and used the rest for validation and testing. As a result, our training data contains sequences out of 5 different machines and our test data contains sequences out of all 14 machines; unseen data from the training machines and

[15] https://keras.io/.

unseen data from the testing machines. The training data was then balanced to contain an equal number of sequences out of healthy and faulty machines.

Results As explained above, we tested the performance of the classifier on two different datasets. The first one is data from the same machines as in the training set and the second is mixed data out of the 5 training machines and the 9 machines not used for training.

Table 2 describes the results for the mixed test data. Note that machines 5 and 11 are the two faulty machines. Part of the data from machine 11 was used for training and the rest for testing. The healthy training data was gathered from machines 2, 4, 6 and 14.

The table shows the true negatives (TN), false negatives (FN), true positives (TP), false positives (FP) and the accuracy of our CNN. From the table it can be seen that all healthy test machines were identified as healthy with an accuracy higher than 0.878, that is with detection rate of more than 87.8% healthy samples. Moreover, we calculated the total precision, recall and accuracy rates over the entire data, gathering results form all machines. These are 0.995, 0.973 and 0.988, respectively. The accuracy rate means that close to 99% of the faulty data was identified as such. Hence, we conclude that the CNN was capable of identifying the faulty test machine based on data from another faulty machine.

In summary, the classification performance of the CNN is very high, with no need for feature engineering. However, the disadvantage of this method for fault detection is the lack of transparency of the results. In a future research we investigate the feature maps which dominate the classification in order to elucidate the underlying classification mechanism.

Method 2: Machine classification based on statistical signal analysis

In this sub-section we perform an analysis using a simpler statistical method than CNNs, which we conducted with the same dataset, yielding a good performance in detecting faulty behavior of machines. In particular, we analyze the time series data of the 14 machines mentioned above in order to identify signatures which are unique for faulty machines and set a threshold for the detection of such faults.

Method We use the same dataset as described above, out of 14 different machines operating under a set of operating conditions. Here we perform a pre-processing of the data in order to select a subset of the operating conditions under which we then perform feature extraction for the construction of health indicators.

After an inspection of the data we observe that the output signal of one of the sensors can be used to construct health indicators for the machine. We denote this signal by S(t). We observe that a subset of process parameters can be pre-selected such that production processes under these conditions can be used for early fault detection before the machine suffers from downtime. We show below that selecting data out of such processes allows us to distinguish faulty from healthy machines. We can then detect these machines as faulty based on cutting processes in which the production outcomes are not yet deteriorated in quality. The pre-selection step of the data is implemented in practice using the query techniques that are mentioned above in "Analysis of query performance in Big Data architecture" section.

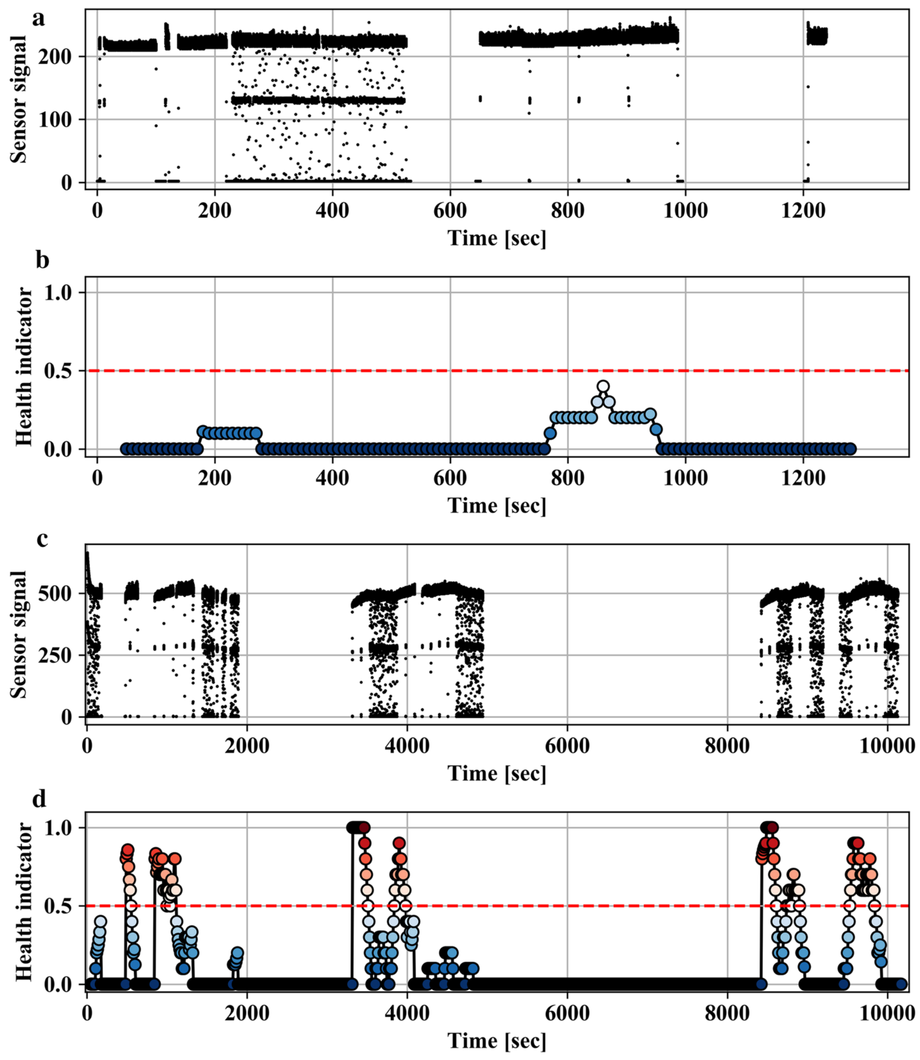


Fig. 9 Fault detection using statistical analysis. **a** Sensor signal of healthy machine. **b** Health indicator for healthy machine. **c** Sensor signal for faulty machine. **d** Health indicator for faulty machine. Red dashed line indicates the detection threshold

The raw signal data S(t) is pre-selected and smoothed. On the smoothed data we run a sliding window robust linear regressor to extract the local slope. Next we perform counting statistics of the slopes within longer time windows, counting the fraction of slopes that exceed a certain slope threshold. This fraction is our health indicator. Using data out of faulty as well as healthy machines we could determine and validate a threshold fraction 0.5 per time window, above which a machine is declared faulty.

The smoothing and windowed operations are again implemented with the help of an efficient query technique as described in "Query processing and optimization" section.

Results: Figure 9 displays two examples of data out of a healthy and a faulty machine. The two upper panels display processes of a healthy machine. Figure 9a shows the raw sensor data and Fig. 9b displays the calculated health indicator as a function of time for this machine. Indeed, during all measurements we observed no time windows with a high fraction of high slopes. Therefore, in this case no fault identification should be issued.

In contrast, Figure 9c, d show the sensor signal and the health indicator for a faulty machine. Here the health indicator, corresponding to the high slope fraction, crosses the threshold level several times during the measurement. This distinction between faulty and healthy machines repeats in all 14 machines that were examined in this experiment. The two faulty machines exhibit time slots with high fractions of high slopes. The detection threshold is therefore crossed repeatedly. In all other machines the threshold was never crossed. Based on the present experiment we conclude that the fraction of high slope data can be used as a health indicator. Note that this feature is independent of the absolute sensor values, but rather reflects their time dependence.

It is worth noting, that we deliberately avoid detecting faults based on the absolute level of the sensor reading $S(t)$. The reason is, that despite the examples shown in the figure, there are cases of high sensor values that are known to be benign, that is, do not lead to reduced production outcomes. Therefore, faulty behavior cannot be detected by merely setting a threshold on the raw sensor reading.

Conclusions We analysed two possible intelligent algorithms for fault detection in the optical system. Both were capable of identifying clearly the two faulty machines out of a set of 14 machines. The first, based on CNN, has the advantage of not relying on a preprocessing step of feature engineering. However, it is hard to interpret and is prone to be sensitive to the selection of training data, especially under variable operating conditions. The second, based on standard statistical analysis, requires the manual extraction of features but is easily interpretable. Moreover, it has the potential to be robust and applicable to a large number of machines under diverse operating conditions. Therefore, in practical applications the second algorithm is often preferred.

Lessons learned and applying best practices

In the preceding proof-of-concept for the fault detection model we were able to verify that our algorithms can learn from data stemming from multiple machines. Moreover, since the input parameter space and the variable domain of operating conditions is huge, it is beneficial for the performance of the algorithms to include data from as many machines as possible for model training. In particular, there is strong evidence, that it is impossible to collect a representative dataset from a single machine in economically meaningful time.

Operating a machine learning model for a complex condition monitoring problem usually includes feature engineering of a vast space of operating conditions throughout the entire asset life. To make it even worse, these operating conditions are partly dependent on the states of contributing subsystems, that deteriorate over time. Last but not least, the components may be replaced from time to time. If those replacements are performed with a new part of identical design, the major effort is to collect properly labeled data. However, if a revised design with a different data signature is put in place, the machine learning models might need to be completely retrained from scratch.

How should these machine learning models be deployed as part of our overall system architecture? In order to enable a short-notice operator alerting system to prevent eminent quality loss in a productive service support system for a fleet of machines, the following tasks are required:

- Periodic re-training and validation of the fault classification model on the entire fleet data.
- Verification of the defined business objectives making use of the precision, recall and accuracy metrics of the algorithms.
- Model deployment in a close to real-time data processing pipeline.

In our scenario, the machine learning models will be deployed and used on the incoming data stream. Upon a positive early fault detection on the continuous data received, the operator of the respective machines will be notified and preventive service measures can be undertaken.

Conclusion

In this paper we presented the design and implementation of an end-to-end Big Data architecture for a real-world use case of intelligent maintenance of a fleet of laser cutting machines.

First, we presented a Big Data architecture that enables both batch and stream processing. In particular, we focused on the important aspect of how to design the system to be resilient to schema changes due to regular software updates of a fleet of laser cutting machines.

Next, we analyzed various physical design choices to optimize query processing in a Big Data architecture. In particular, we studied the impact of various data partitioning strategies as well as Z-ordering on the performance of multi-dimensional query processing. The results show that data partitioning using Delta Lakes with a modest number of partitions is often the best strategy. The results also demonstrate that Z-ordering typically does not perform better than data partitioning. However, Z-ordering is a decent alternative design choice for high-cardinality attributes where data partitioning would break due to a high number of physical files.

Finally, we evaluated two different approaches for a specific use case of health monitoring. The first approach uses machine learning techniques based on convolutional neural networks, while the second approach uses standard statistical and signal processing methods. Our results demonstrate that standard statistical methods are the better design choice for our use case due to their robustness towards variable operating conditions.

Abbreviations

CERN: European organization for nuclear research; CNN: Convolutional neural network; CPU: Central processing unit; CTI: Swiss commission for technology and innovation; FN: False negative; FP: False positive; FPGA: Field programmable gate array; GB: Gigabyte; IoT: Internet of things; JSON: JavaScript object notation; ML: Machine learning; PQ: Point query; RQ: Range query; RUL: Remaining useful life; TN: True negative; TP: True positive.

Acknowledgements

Not applicable.

Authors' contributions

TH and GS provided the use case and the domain expertise. KS and AS designed the overall Big Data system architecture. AS and CL implemented the system and performed the experiments on query processing. LGH performed the data analytics. All authors read and approved the final manuscript.

Funding

The work was supported by the Swiss Commission for Technology and Innovation (CTI) under Grant 25728.1 PFES-ES.

Availability of data and materials

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Author details

[1] Zurich University of Applied Sciences, Obere Kirchgasse 2, 8400 Winterthur, Switzerland. [2] Bystronic Laser AG, Industriestrasse 21, 3362 Niederönz, Switzerland.

Received: 15 April 2020 Accepted: 30 July 2020
Published online: 12 August 2020

References

1. Sima A-C, Stockinger K, Affolter K, Braschler M, Monte P, Kaiser L. A hybrid approach for alarm verification using stream processing, machine learning and text analytics. In: EDBT 2018, Vienna, Austria, 26-29 March 2018. ACM; 2018.
2. Lee J, Ardakani HD, Yang S, Bagheri B. Industrial big data analytics and cyber-physical systems for future maintenance & service innovation. Procedia Cirp. 2015;38:3–7.
3. Canizo M, Onieva E, Conde A, Charramendieta S, Trujillo S. Real-time predictive maintenance for wind turbines using big data frameworks. In: 2017 IEEE International Conference on Prognostics and Health Management (ICPHM). IEEE; 2017. p. 70–7.
4. Syafrudin M, Alfian G, Fitriyani NL, Rhee J. Performance analysis of iot-based sensor, big data processing, and machine learning model for real-time monitoring system in automotive manufacturing. Sensors. 2018;18(9):2946.
5. Comer D. Ubiquitous b-tree. ACM Comput Surv. 1979;11(2):121–37.
6. Graefe G. Query evaluation techniques for large databases. ACM Comput Surv. 1993;25(2):73–169.
7. Finkel RA, Bentley JL. Quad trees a data structure for retrieval on composite keys. Acta Informatica. 1974;4(1):1–9.
8. Bentley JL. Multidimensional binary search trees used for associative searching. Commun ACM. 1975;18(9):509–17.
9. Guttman A. R-trees: A dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, 1984, pp. 47–57.
10. Hilbert D. Ueber die reellen züge algebraischer curven. Mathematische Annalen. 1891;38(1):115–38.
11. Morton GM. A computer oriented geodetic data base and a new technique in file sequencing, 1966.
12. Spiegler I, Maayan R. Storage and retrieval considerations of binary data bases. Inform Process Manag. 1985;21(3):233–54.
13. Wu K, Shoshani A, Stockinger K. Analyses of multi-level and multi-component compressed bitmap indexes. ACM Trans Database Syst. 2008;35(1):1–52.
14. Nathan V, Ding J, Alizadeh M, Kraska T. Learning Multi-dimensional Indexes 2019. arxiv:1912.01668.
15. MongoDB Documentation: Indexes. https://docs.mongodb.com/manual/indexes/. Accessed 13 Feb 2020.
16. Neo4j Documentation: Index configuration. https://neo4j.com/docs/operations-manual/current/performance/index-configuration/#index-configuration-btree. Accessed 13 Feb 2020.
17. Apache Hive Confluence: LanguageManual Indexing. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Indexing. Accessed 13 Feb 2020.
18. PostgreSQL Documentation: Combining Multiple Indexes. https://www.postgresql.org/docs/10/indexes-bitmap-scans.html. Accessed 13 Feb 2020.
19. Stockinger K, Bödi R, Heitz J, Weinmann T. Zns-efficient query processing with zurichnosql. Data Knowl Eng. 2017;112:38–54.
20. Amazon AWS: Amazon Redshift Engineering's Advanced Table Design Playbook: Compound and Interleaved Sort Keys. https://aws.amazon.com/de/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-compound-and-interleaved-sort-keys. Accessed 30 Feb 2020.
21. Amazon AWS Database Blog: Z-Order Indexing for Multifaceted Queries in Amazon DynamoDB: Part 1. https://aws.amazon.com/de/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-1. Accessed 30 Jan 2020.
22. Databricks Engineering Blog: Optimize Performance with File Management. https://docs.databricks.com/delta/optimizations/file-mgmt.html. Accessed 28 Nov 2019.
23. Khan S, Yairi T. A review on the application of deep learning in system health management. Mech Syst Sign Process. 2018;107:241–65.
24. Jing L, Zhao M, Li P, Xu X. A convolutional neural network based feature learning and fault diagnosis method for the condition monitoring of gearbox. Measurement. 2017;111:1–10.
25. Chen Z, Li C, Sanchez R-V. Gearbox fault identification and classification with convolutional neural networks. Shock Vibr. 2015;2015.
26. Zhang W, Li C, Peng G, Chen Y, Zhang Z. A deep convolutional neural network with new training methods for bearing fault diagnosis under noisy environment and different working load. Mech Syst Sign Process. 2018;100:439–53.

27. Liu C-L, Hsaio W-H, Tu Y-C. Time series classification with multivariate convolutional neural network. IEEE Trans Ind Electr. 2018;66(6):4788–97.
28. Bellatreche L, Boukhalfa K, Richard P. Data partitioning in data warehouses: Hardness study, heuristics and oracle validation. In: International Conference on Data Warehousing and Knowledge Discovery. Springer: New York; 2008. pp. 87–96.
29. Stockinger K, Wu K. Bitmap indices for data warehouses. In: Data Warehouses and OLAP: Concepts, Architectures and Solutions. IGI Global, 2007. p. 157–78.
30. Ionescu A. Processing Petabytes of Data in Seconds with Databricks Delta. https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html. Accessed 28 Nov 2019.
31. Bai S, Kolter JZ, Koltun V. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. arXiv preprint arXiv:1803.01271 2018.

Publisher's Note