Open for Innovation
KNIME

Fast Tables:
The Columnar Table Backend

Marc Bux
November 20, 2020

#KNIMESummit

Fast Tables

- Complete rewrite of KNIME‘s table storage backend
- To be released in Labs with AP 4.3

Fast Tables

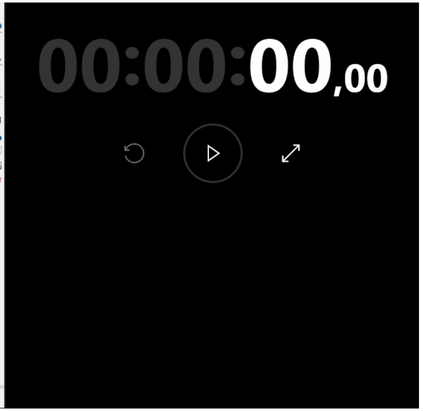
- Complete rewrite of KNIME‘s table storage backend
- To be released in Labs with AP 4.3
- Preliminary Benchmark
  - Some frequently used nodes
  - 2 GB Heap (-Xmx), 2 GB Off-Heap

Fast Tables

- Complete rewrite of KNIME‘s table storage backend
- To be released in Labs with AP 4.3
- Preliminary Benchmark
  - Some frequently used nodes
  - 2 GB Heap (-Xmx), 2 GB Off-Heap

You are using a nightly build!

83M of 244M



Agenda

1. A New Table Backend for Improved Performance
2. Design Decisions and Benefits
3. Cache Architecture and Configuration
4. New Table API

Agenda

1. A New Table Backend for Improved Performance
2. Design Decisions and Benefits
3. Cache Architecture and Configuration
4. New Table API

A Little History

- In KNIME 3.5 and earlier, all tables with more than 100k cells were synchronously persisted to disk

A Little History

- In KNIME 3.5 and earlier, all tables with more than 100k cells were synchronously persisted to disk

- KNIME 3.6 introduced the Parquet Column Store
  - Major speedups when accessing only parts of a table (selected columns or a range of rows)
  - Lesson learned:
    It is the fact that we persist to disk (and not how we persist to disk) that costs performance

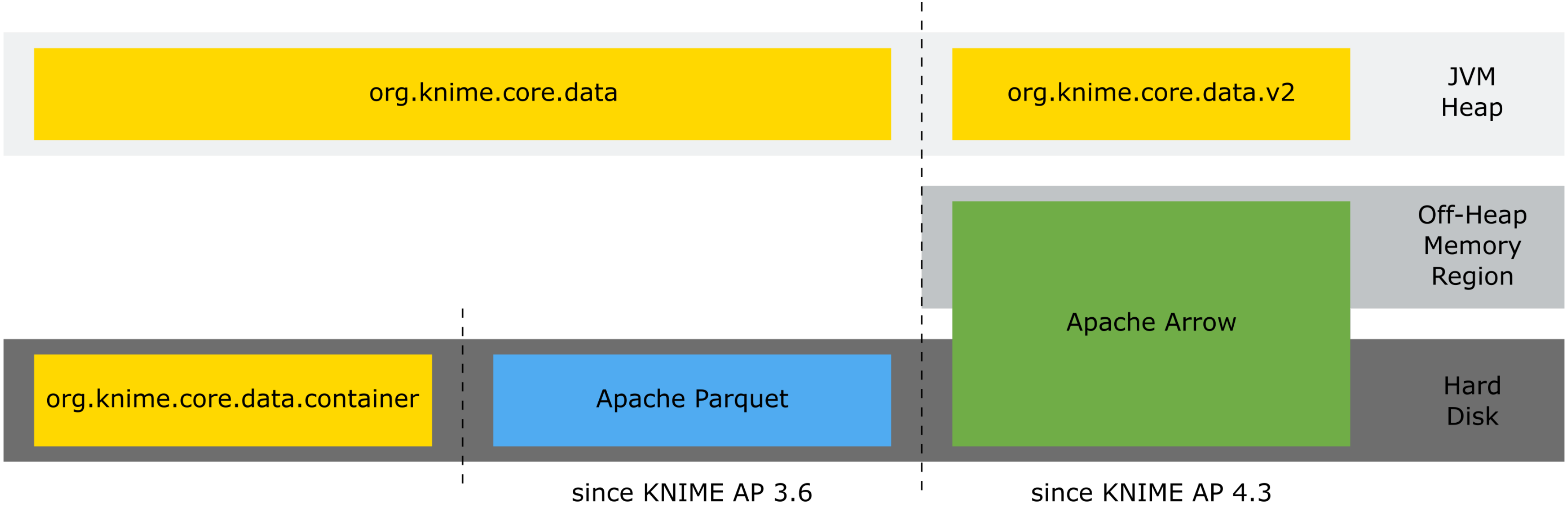
A Little History

- In KNIME 3.5 and earlier, all tables with more than 100k cells were synchronously persisted to disk

- KNIME 3.6 introduced the Parquet Column Store
  - Major speedups when accessing only parts of a table (selected columns or a range of rows)
  - Lesson learned:
    It is the fact that we persist to disk (and not how we persist to disk) that costs performance

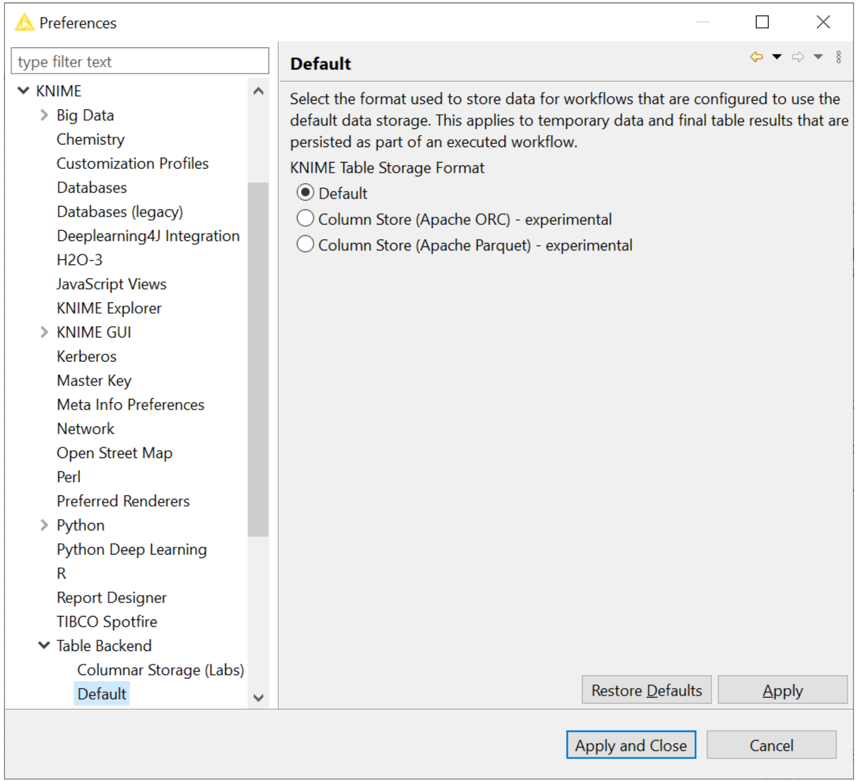
- KNIME 4.0 introduced an LRU cache for medium-sized tables
  - Medium-sized: larger than 5k cells, but not too large to fit into the heap
  - Lesson learned:
    Keeping tables in the JVM heap for a long time puts a lot of pressure on garbage collection

A New Table Backend



- Apache Parquet / ORC: “column-oriented data storage formats of the Apache Hadoop ecosystem”
- Apache Arrow: “language-agnostic software framework for developing data analytics applications that process columnar data”

Preference Page: Old Versus New

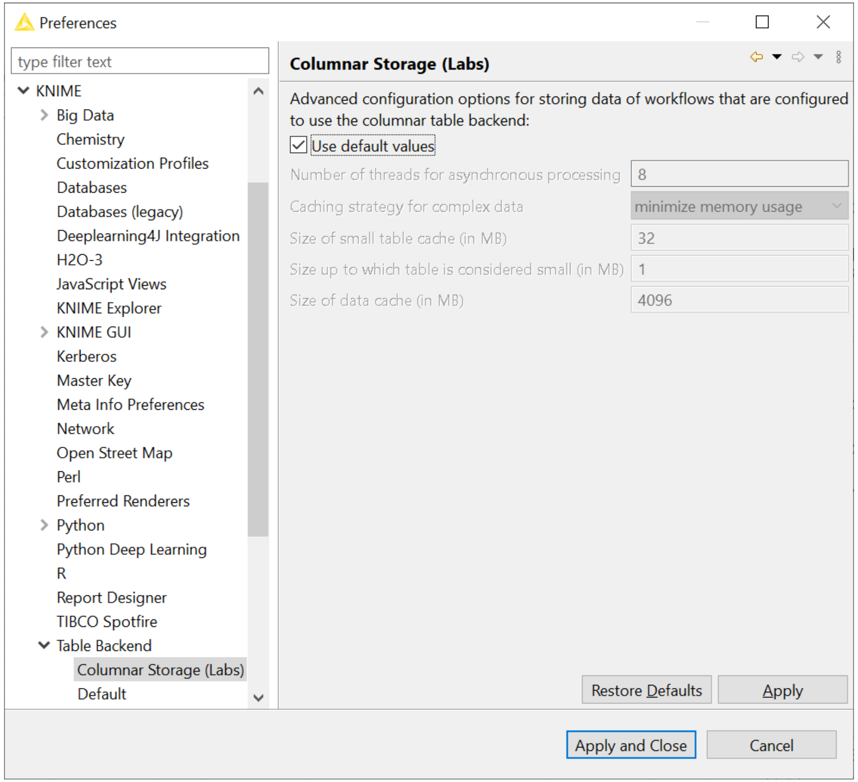


Default

Select the format used to store data for workflows that are configured to use the default data storage. This applies to temporary data and final table results that are persisted as part of an executed workflow.

KNIME Table Storage Format

- ● Default
- ○ Column Store (Apache ORC) - experimental
- ○ Column Store (Apache Parquet) - experimental



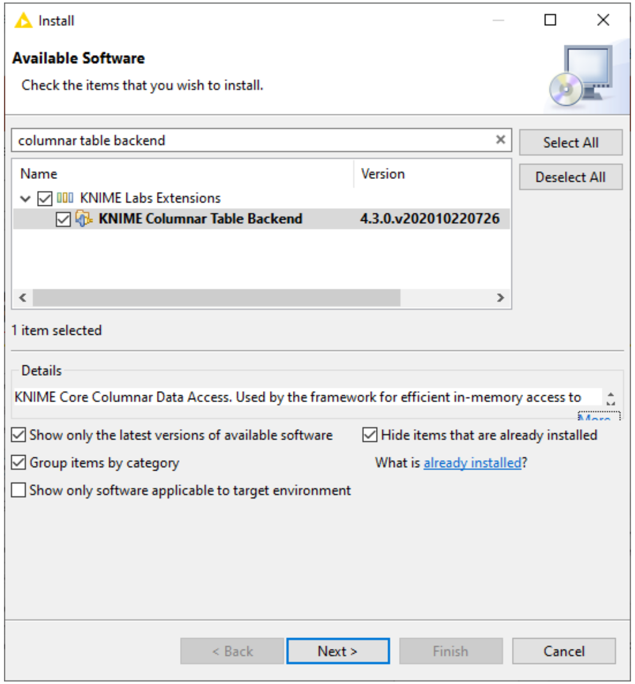
Columnar Storage (Labs)

Advanced configuration options for storing data of workflows that are configured to use the columnar table backend:

☑ Use default values

Number of threads for asynchronous processing	8
Caching strategy for complex data	minimize memory usage
Size of small table cache (in MB)	32
Size up to which table is considered small (in MB)	1
Size of data cache (in MB)	4096

Getting Started



Available Software

Check the items that you wish to install.

columnar table backend

Name	Version
☑ KNIME Labs Extensions	
☑ **KNIME Columnar Table Backend**	**4.3.0.v202010220726**

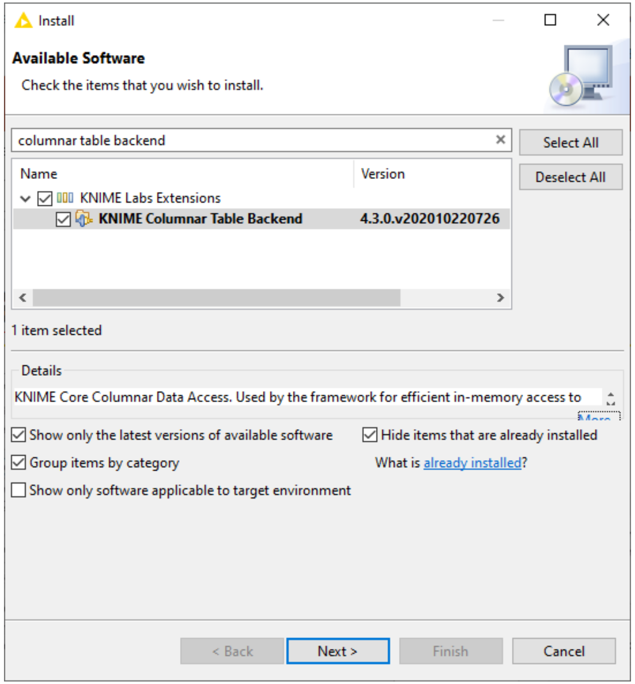
1 item selected

Details

KNIME Core Columnar Data Access. Used by the framework for efficient in-memory access to

☑ Show only the latest versions of available software
☑ Hide items that are already installed
☑ Group items by category
What is already installed?
☐ Show only software applicable to target environment

Getting Started



Install

Available Software

Check the items that you wish to install.

columnar table backend

Name	Version
☑ KNIME Labs Extensions	
☑ **KNIME Columnar Table Backend**	**4.3.0.v202010220726**

1 item selected

Details

KNIME Core Columnar Data Access. Used by the framework for efficient in-memory access to

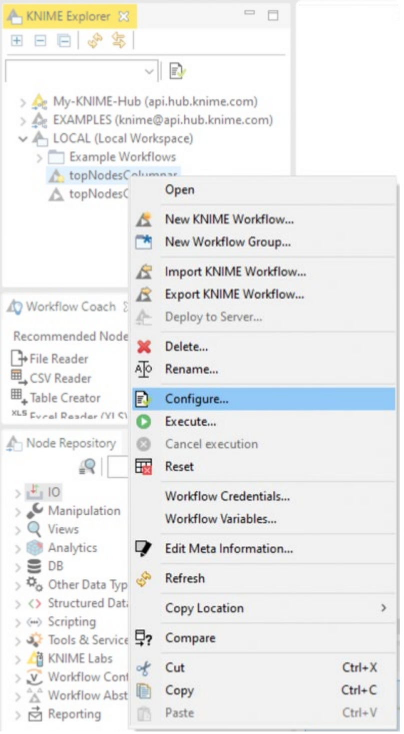
☑ Show only the latest versions of available software

☑ Hide items that are already installed

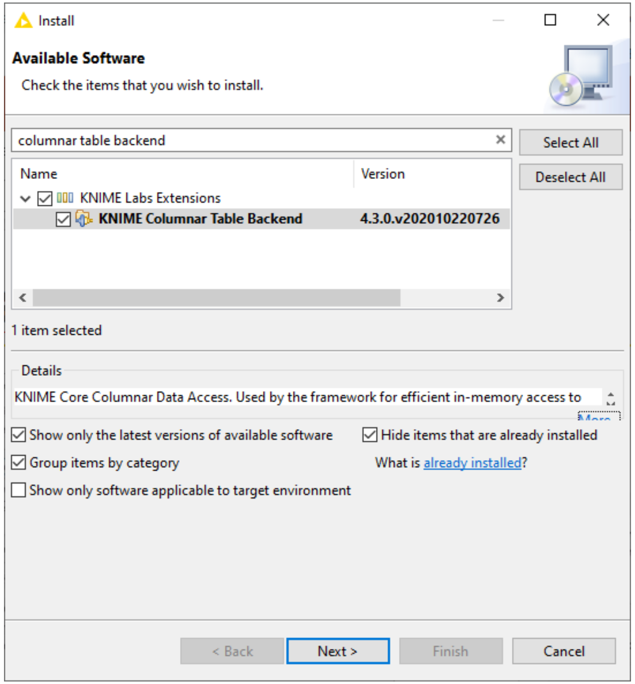
☑ Group items by category

What is already installed?

☐ Show only software applicable to target environment



Getting Started



Available Software

Check the items that you wish to install.

columnar table backend

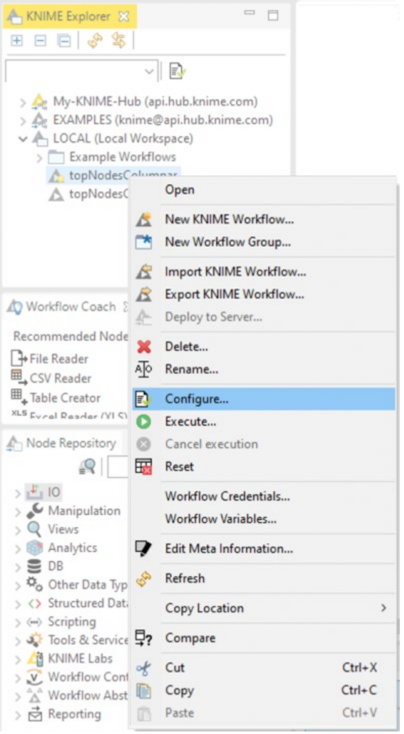
Name	Version
☑ KNIME Labs Extensions	
☑ **KNIME Columnar Table Backend**	**4.3.0.v202010220726**

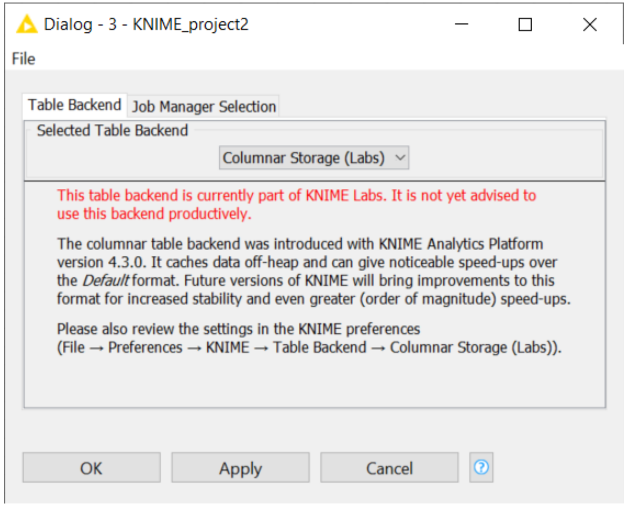
1 item selected

Details

KNIME Core Columnar Data Access. Used by the framework for efficient in-memory access to

☑ Show only the latest versions of available software
☑ Hide items that are already installed
☑ Group items by category
What is already installed?
☐ Show only software applicable to target environment





Selected Table Backend: Columnar Storage (Labs)

This table backend is currently part of KNIME Labs. It is not yet advised to use this backend productively.

The columnar table backend was introduced with KNIME Analytics Platform version 4.3.0. It caches data off-heap and can give noticeable speed-ups over the *Default* format. Future versions of KNIME will bring improvements to this format for increased stability and even greater (order of magnitude) speed-ups.

Please also review the settings in the KNIME preferences (File → Preferences → KNIME → Table Backend → Columnar Storage (Labs)).

Agenda

1. A New Table Backend for Improved Performance
2. Design Decisions and Benefits
3. Cache Architecture and Configuration
4. New Table API

Agenda

1. A New Table Backend for Improved Performance
2. Design Decisions and Benefits
3. Cache Architecture and Configuration
4. New Table API

Keep Primitive Data Primitive

- On 64-bit platforms with more than 32 GB heap space:
  - Size of object header: 16 bytes
  - Size of object reference: 8 bytes
  - Total object size padded to a multiple of 8 bytes

Keep Primitive Data Primitive

- On 64-bit platforms with more than 32 GB heap space:
  - Size of object header: 16 bytes
  - Size of object reference: 8 bytes
  - Total object size padded to a multiple of 8 bytes
- IntCell (single int field)
  - 24 bytes in memory, but information can be represented with 4 bytes

Keep Primitive Data Primitive

- On 64-bit platforms with more than 32 GB heap space:
  - Size of object header: 16 bytes
  - Size of object reference: 8 bytes
  - Total object size padded to a multiple of 8 bytes
- IntCell (single int field)
  - 24 bytes in memory, but information can be represented with 4 bytes
- LocalDateTimeCell (LocalDateTime and LocalTime fields)
  - 80 bytes in memory, but information can be represented with 16 bytes

Keep Primitive Data Primitive

- On 64-bit platforms with more than 32 GB heap space:
  - Size of object header: 16 bytes
  - Size of object reference: 8 bytes
  - Total object size padded to a multiple of 8 bytes
- IntCell (single int field)
  - 24 bytes in memory, but information can be represented with 4 bytes
- LocalDateTimeCell (LocalDateTime and LocalTime fields)
  - 80 bytes in memory, but information can be represented with 16 bytes

\+ Reduce memory footprint, hold more data in memory

\+ Less object creations, less work for garbage collector

JVM Heap and Garbage Collection

- JVM Heap
  - Region of memory for dynamic memory allocation for Java objects
  - Size controllable via -Xmx parameter

JVM Heap and Garbage Collection

- JVM Heap
  - Region of memory for dynamic memory allocation for Java objects
  - Size controllable via -Xmx parameter
- Garbage Collector
  - Periodically iterates over heap and frees memory occupied by unreferenced objects

Cache Long-Living in-Memory Data Off-Heap

- JVM Heap
  - Region of memory for dynamic memory allocation for Java objects
  - Size controllable via -Xmx parameter
- Garbage Collector
  - Periodically iterates over heap and frees memory occupied by unreferenced objects

− Non-primitive data must be serialized (converted to byte-array)

− More susceptible to memory leaks

Cache Long-Living in-Memory Data Off-Heap

- JVM Heap
  - Region of memory for dynamic memory allocation for Java objects
  - Size controllable via -Xmx parameter
- Garbage Collector
  - Periodically iterates over heap and frees memory occupied by unreferenced objects

− Non-primitive data must be serialized (converted to byte-array)

− More susceptible to memory leaks

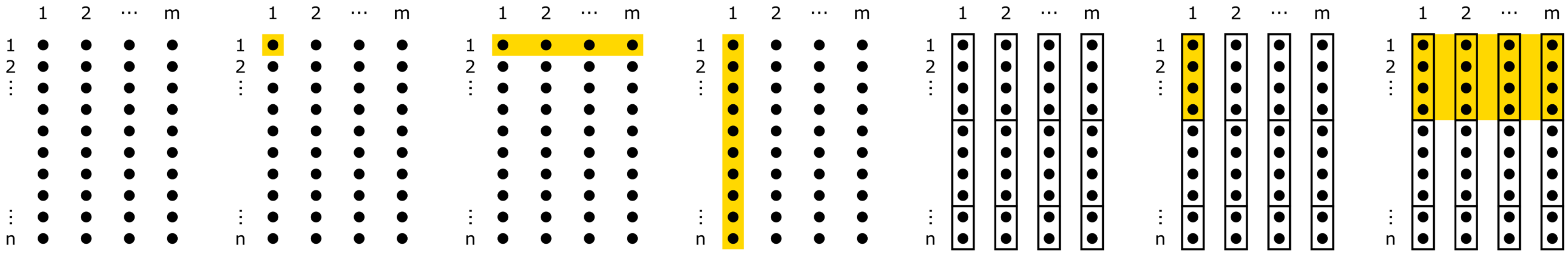
\+ Take load off garbage collection

\+ More controllable memory footprint

\+ Reduce interference between caching of data and node operations

\+ First step in enabling shared memory with other languages (e.g., Python)

Table Structure & Other Benefits



cell

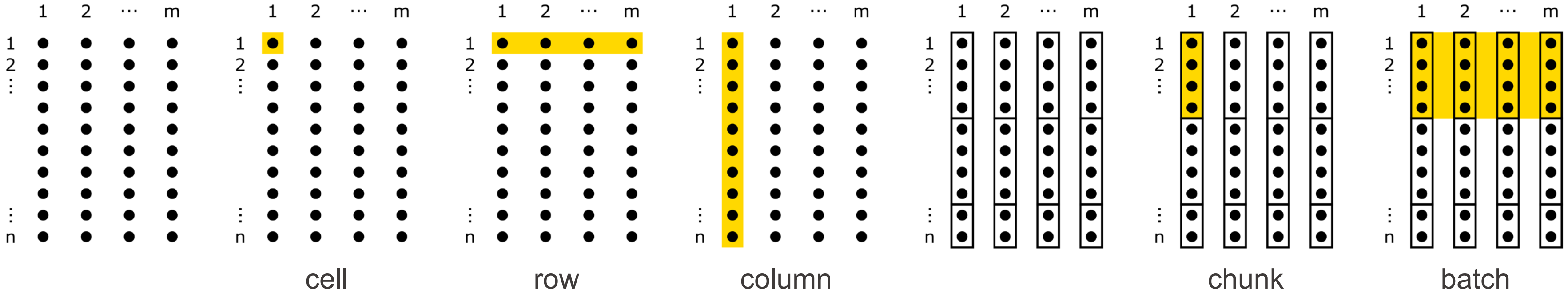
row

column

chunk

batch

Table Structure & Other Benefits



- \+ (A lot) less operations per cell, more operations per chunk
- \+ Cache chunks, as opposed to full tables
- \+ Cleaner, more maintainable code base

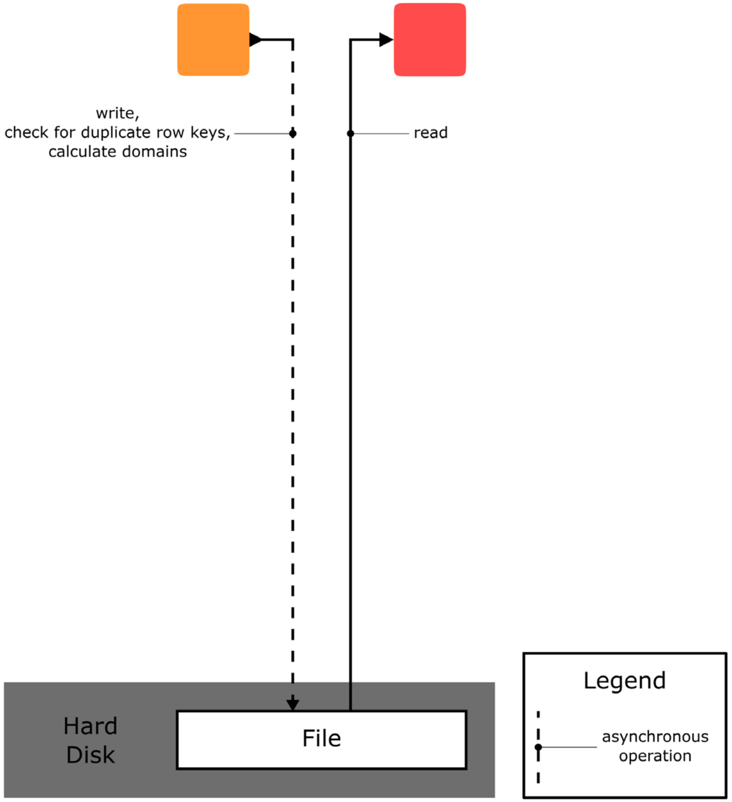
Agenda

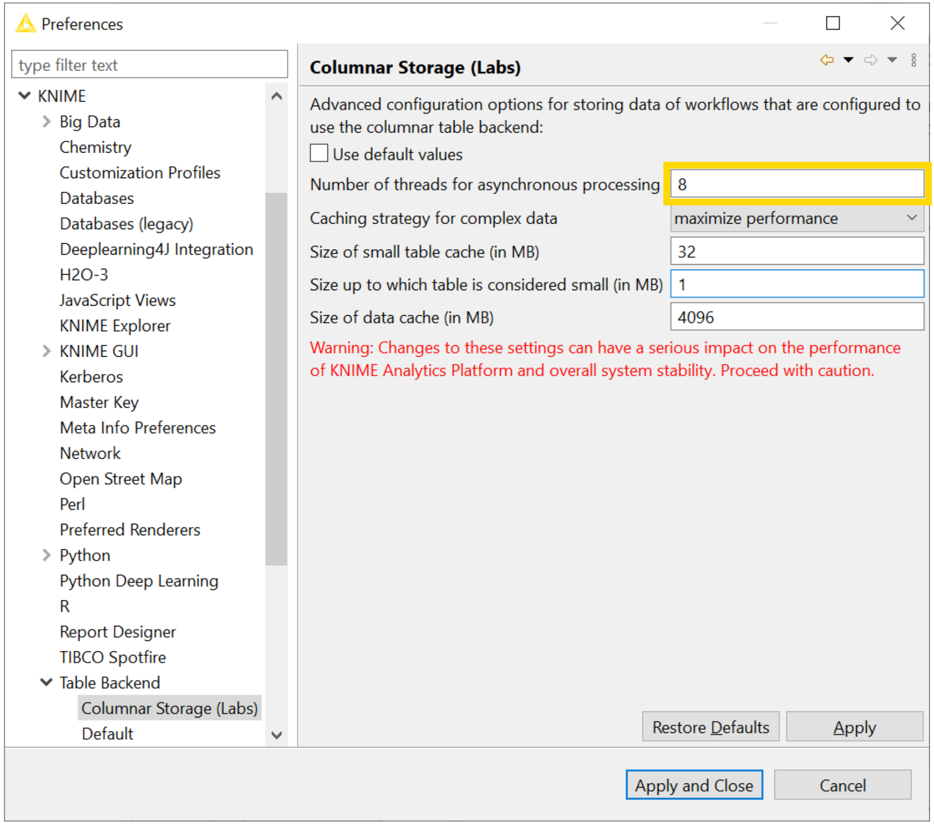
1. A New Table Backend for Improved Performance
2. Design Decisions and Benefits
3. Cache Architecture and Configuration
4. New Table API

Agenda

1. A New Table Backend for Improved Performance
2. Design Decisions and Benefits
3. Cache Architecture and Configuration
4. New Table API

Cache Architecture & Configuration





Columnar Storage (Labs)

Advanced configuration options for storing data of workflows that are configured to use the columnar table backend:

☐ Use default values

Number of threads for asynchronous processing: 8

Caching strategy for complex data: maximize performance

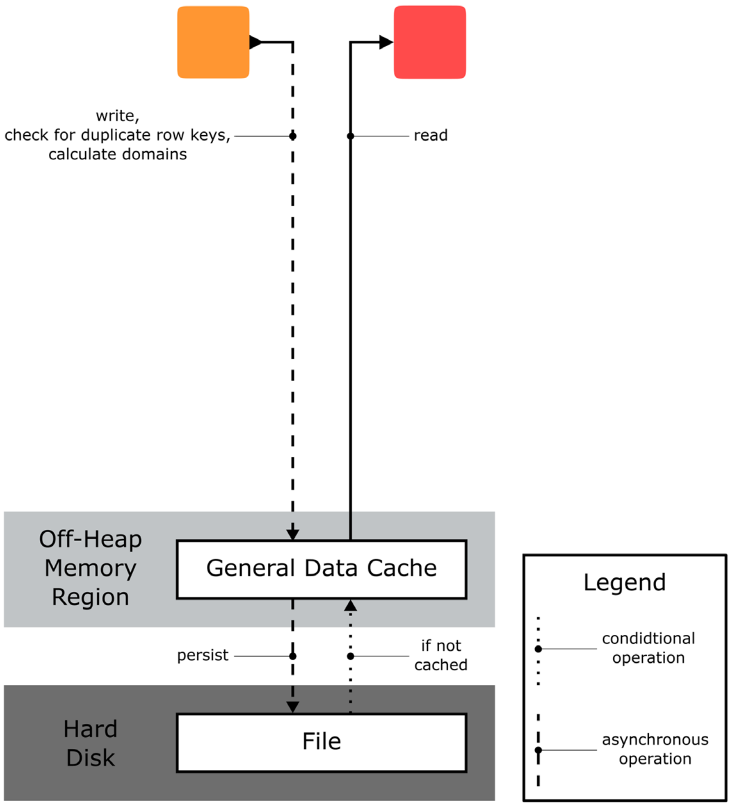
Size of small table cache (in MB): 32

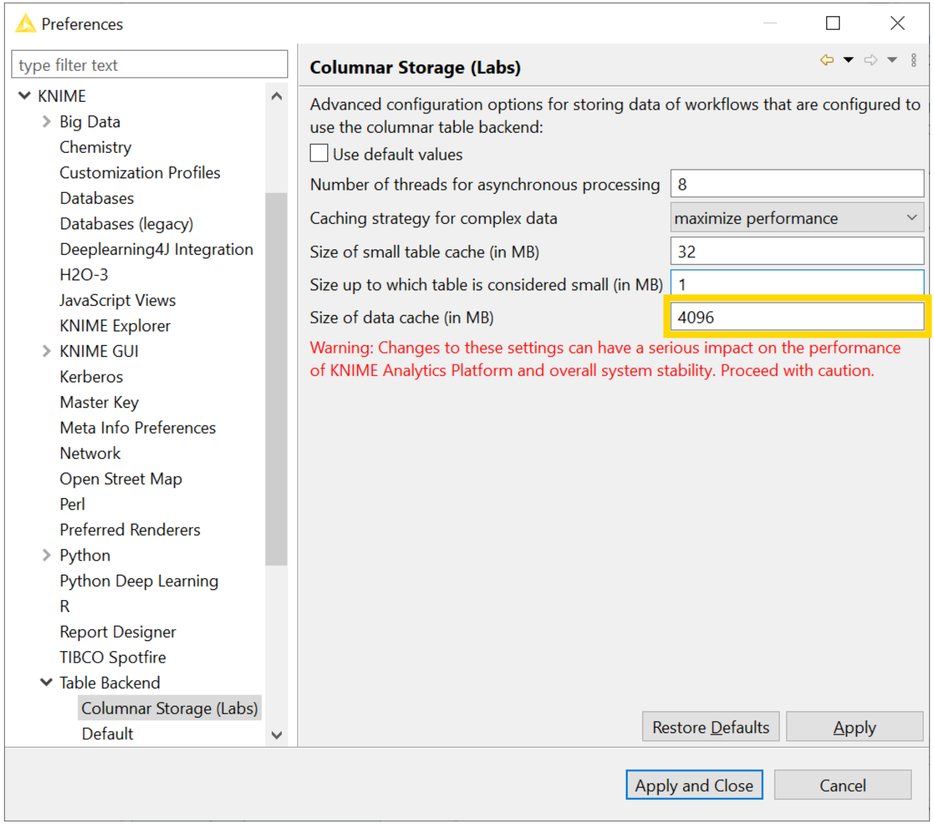
Size up to which table is considered small (in MB): 1

Size of data cache (in MB): 4096

Warning: Changes to these settings can have a serious impact on the performance of KNIME Analytics Platform and overall system stability. Proceed with caution.

Cache Architecture & Configuration





Columnar Storage (Labs)

Advanced configuration options for storing data of workflows that are configured to use the columnar table backend:

☐ Use default values

Number of threads for asynchronous processing: 8

Caching strategy for complex data: maximize performance

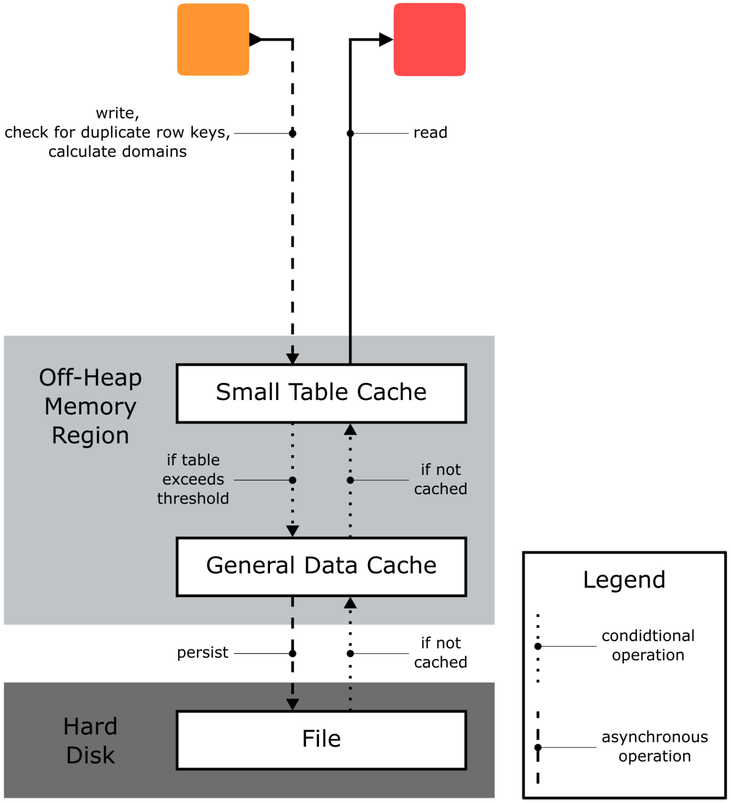
Size of small table cache (in MB): 32

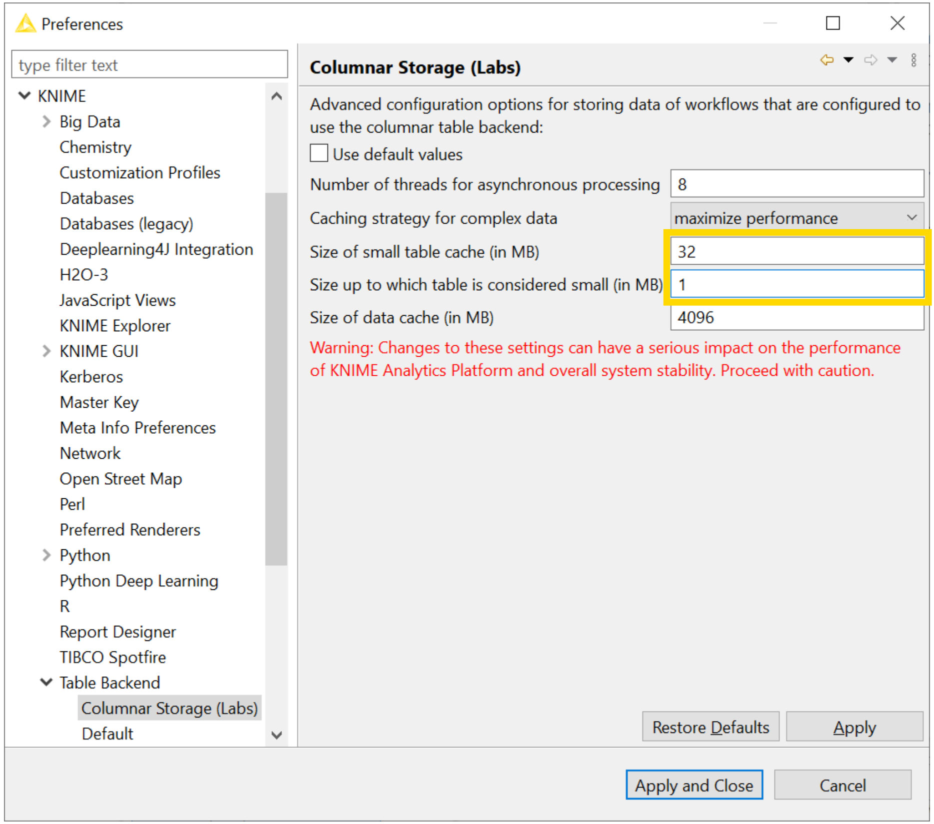
Size up to which table is considered small (in MB): 1

Size of data cache (in MB): 4096

Warning: Changes to these settings can have a serious impact on the performance of KNIME Analytics Platform and overall system stability. Proceed with caution.

Cache Architecture & Configuration





Columnar Storage (Labs)

Advanced configuration options for storing data of workflows that are configured to use the columnar table backend:

☐ Use default values

Number of threads for asynchronous processing: 8

Caching strategy for complex data: maximize performance

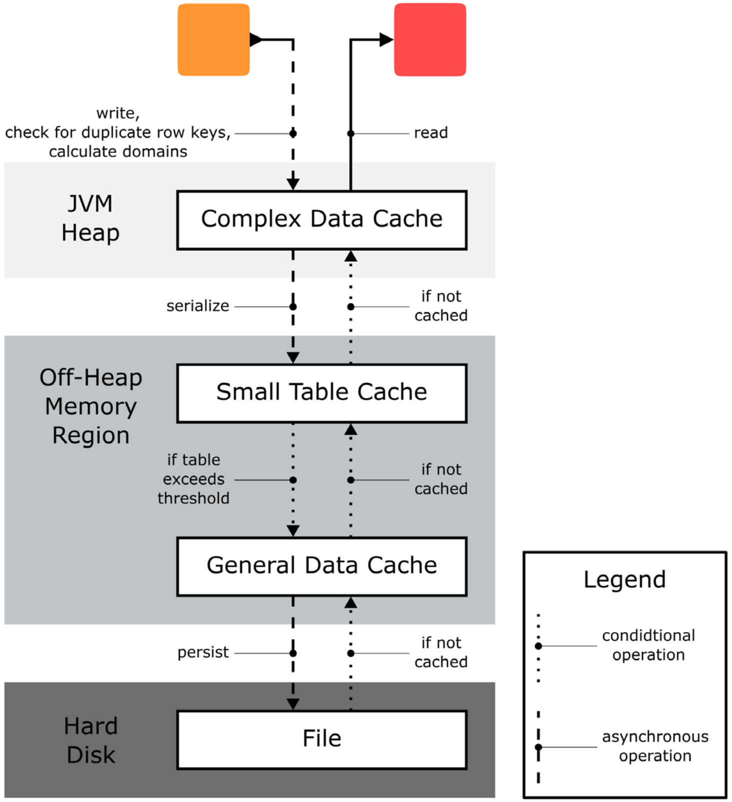
Size of small table cache (in MB): 32

Size up to which table is considered small (in MB): 1

Size of data cache (in MB): 4096

Warning: Changes to these settings can have a serious impact on the performance of KNIME Analytics Platform and overall system stability. Proceed with caution.

Cache Architecture & Configuration



write,
check for duplicate row keys,
calculate domains

read

JVM Heap

Complex Data Cache

serialize

if not cached

Off-Heap Memory Region

Small Table Cache

if table exceeds threshold

if not cached

General Data Cache

persist

if not cached

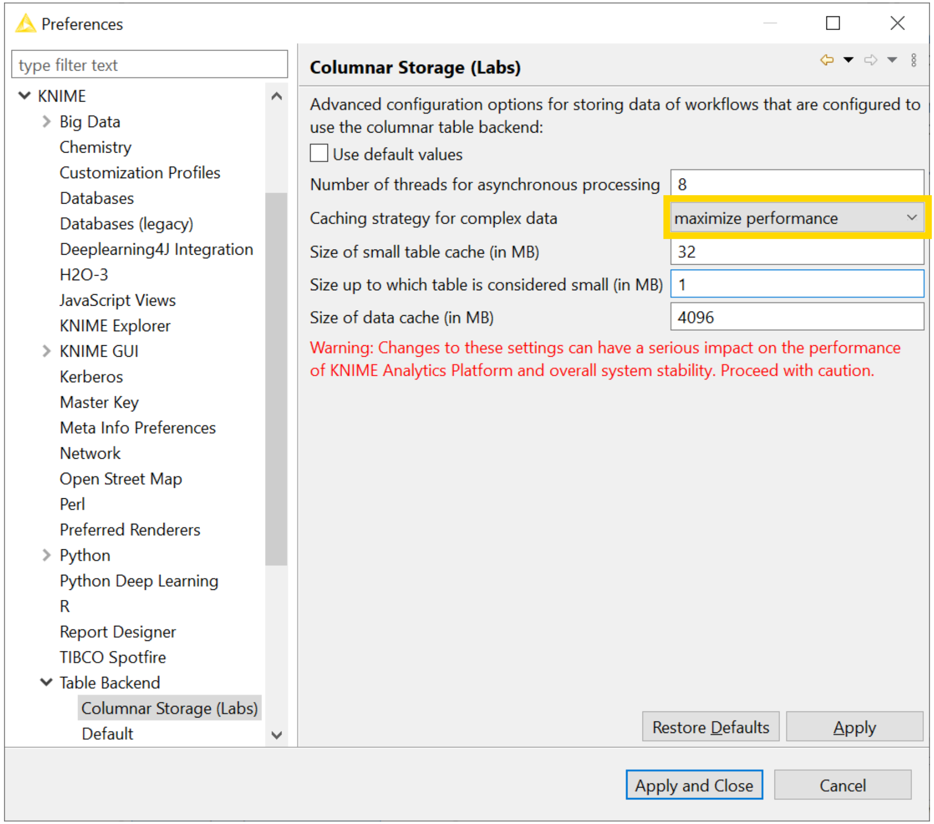
Hard Disk

File

Legend

condidtional operation

asynchronous operation



Columnar Storage (Labs)

Advanced configuration options for storing data of workflows that are configured to use the columnar table backend:

☐ Use default values

Number of threads for asynchronous processing	8
Caching strategy for complex data	maximize performance
Size of small table cache (in MB)	32
Size up to which table is considered small (in MB)	1
Size of data cache (in MB)	4096

Warning: Changes to these settings can have a serious impact on the performance of KNIME Analytics Platform and overall system stability. Proceed with caution.

Agenda

1. A New Table Backend for Improved Performance
2. Design Decisions and Benefits
3. Cache Architecture and Configuration
4. New Table API

Agenda

1. A New Table Backend for Improved Performance
2. Design Decisions and Benefits
3. Cache Architecture and Configuration
4. New Table API

Minimal Data Source (subject to change)

```java
protected BufferedDataTable[] execute(final BufferedDataTable[] inData, final ExecutionContext exec) throws Exception {
    final DataTableSpec spec = new DataTableSpec(new DataColumnSpecCreator("Column 0", IntCell.TYPE).createSpec());
    final BufferedDataContainer container = exec.createDataContainer(spec);
    for (int i = 0; i < 1000; i++) {
        final RowKey rowKey = RowKey.createRowKey((long)i);
        final DataCell cell = new IntCell(i);
        final DataRow row = new DefaultRow(rowKey, cell);
        container.addRowToTable(row);
    }
    container.close();
    return new BufferedDataTable[]{container.getTable()};
}
```

Minimal Data Source (subject to change)

```java
protected BufferedDataTable[] execute(final BufferedDataTable[] inData, final ExecutionContext exec) throws Exception {
    final DataTableSpec spec = new DataTableSpec(new DataColumnSpecCreator("Column 0", IntCell.TYPE).createSpec());
    final BufferedDataContainer container = exec.createDataContainer(spec);
    for (int i = 0; i < 1000; i++) {
        final RowKey rowKey = RowKey.createRowKey((long)i);
        final DataCell cell = new IntCell(i);
        final DataRow row = new DefaultRow(rowKey, cell);
        container.addRowToTable(row);
    }
    container.close();
    return new BufferedDataTable[]{container.getTable()};
}
```

Minimal Data Source (subject to change)

```
protected BufferedDataTable[] execute(final BufferedDataTable[] inData, final ExecutionContext exec) throws Exception {
    final DataTableSpec spec = new DataTableSpec(new DataColumnSpecCreator("Column 0", IntCell.TYPE).createSpec());
    final BufferedDataContainer container = exec.createDataContainer(spec);
    for (int i = 0; i < 1000; i++) {
        final RowKey rowKey = RowKey.createRowKey((long)i);
        final DataCell cell = new IntCell(i);
        final DataRow row = new DefaultRow(rowKey, cell);
        container.addRowToTable(row);
    }
    container.close();
    return new BufferedDataTable[]{container.getTable()};
}
```

```
protected BufferedDataTable[] execute(final BufferedDataTable[] inData, final ExecutionContext exec) throws Exception {
    final DataTableSpec spec = new DataTableSpec(new DataColumnSpecCreator("Column 0", IntCell.TYPE).createSpec());
    try (final RowContainer container = exec.createRowContainer(spec);
         final RowWriteCursor cursor = container.createCursor()) { // multiple cursors in the future?
        for (int i = 0; i < 1000; i++) {
            final RowWrite row = cursor.forward();
            final String rowKey = String.format("Row%d", i); // auto row keys in the future?
            row.setRowKey(rowKey);
            final IntWriteValue value = row.getWriteValue(0);
            value.setIntValue(i);
        }
        return new BufferedDataTable[]{container.finish()};
    }
}
```

Minimal Data Sink (subject to change)

```java
protected BufferedDataTable[] execute(final BufferedDataTable[] inData, final ExecutionContext exec) throws Exception {
    try (final CloseableRowIterator iterator = inData[0].iterator()) {
        while (iterator.hasNext()) {
            final DataRow row = iterator.next();
            final DataCell cell = row.getCell(0);
            final IntValue value = (IntValue)cell;
            final int i = value.getIntValue(); // do something with i
        }
    }
    return new BufferedDataTable[0];
}
```

Minimal Data Sink (subject to change)

```
protected BufferedDataTable[] execute(final BufferedDataTable[] inData, final ExecutionContext exec) throws Exception {
    try (final CloseableRowIterator iterator = inData[0].iterator()) {
        while (iterator.hasNext()) {
            final DataRow row = iterator.next();
            final DataCell cell = row.getCell(0);
            final IntValue value = (IntValue)cell;
            final int i = value.getIntValue(); // do something with i
        }
    }
    return new BufferedDataTable[0];
}
```

Minimal Data Sink (subject to change)

```java
protected BufferedDataTable[] execute(final BufferedDataTable[] inData, final ExecutionContext exec) throws Exception {
    try (final CloseableRowIterator iterator = inData[0].iterator()) {
        while (iterator.hasNext()) {
            final DataRow row = iterator.next();
            final DataCell cell = row.getCell(0);
            final IntValue value = (IntValue)cell;
            final int i = value.getIntValue(); // do something with i
        }
    }
    return new BufferedDataTable[0];
}
```

```java
protected BufferedDataTable[] execute(final BufferedDataTable[] inData, final ExecutionContext exec) throws Exception {
    try (final RowCursor cursor = inData[0].cursor()) {
        while (cursor.canForward()) {
            final RowRead row = cursor.forward();
            final IntValue value = row.getValue(0);
            final int i = value.getIntValue(); // do something with i
        }
    }
    return new BufferedDataTable[0];
}
```

Agenda

1. A New Table Backend for Improved Performance
2. Design Decisions and Benefits
3. Cache Architecture and Configuration
4. New Table API

Summary

1. A New Table Backend for Improved Performance
2. Design Decisions and Benefits
3. Cache Architecture and Configuration
4. New Table API

Outlook

- The feature is in a fully usable (Labs) state
- Fully compatible with all nodes and data through a transition layer
- Please, use it now and share feedback with us

Outlook

- The feature is in a fully usable (Labs) state
- Fully compatible with all nodes and data through a transition layer
- Please, use it now and share feedback with us

- Next steps after the AP 4.3 release:
  1. Additional improvements to become production-ready and squeeze out more performance gains
  2. Make use of backend in streaming
  3. Rewrite frequently used nodes to use new table API for yet more performance improvements
  4. Review other places in org.knime.core where we currently loose performance