InfoDesign GmbH

Development for Customers

Benutzerhandbuch IM4-Archiver

Stand: 1.5.1 7/1/2022

Allgemeine Automationsverfahren

IM4-Archiver 1.5.1

Gedruckt: July 2022

Herausgeber

InfoDesign GmbH

Großes Feld 23
25421 Pinneberg

Tel +49 4101 - 69 31 - 54
Fax +49 4101 - 69 31 - 56

Mail mail@infodesign.de

Support
Tel +49 4101 595 8991
Mail support@infodesign.de

www.infodesign.de

Inhaltsverzeichnis

1 Einleitung

Beim Archivieren und Austauschen von strukturierten Daten aus Datenbanksystemen kann die Problematik auftreten, dass durch technologischen Wandel, wie z.B. Updates des Datenbanksystems oder der Wechsel in ein anderes Datenbank- oder Betriebssystem, ein Archiv nicht mehr korrekt gelesen und wiederhergestellt werden kann.

IM4-Archiver löst diese Problematik, indem die Metadaten und Daten der Schemas und Tabellen einer Datenbank in einem datenbank-unabhängigen Format gespeichert werden. Das Format der Wahl ist JSON. Da es sich bei JSON um ein reines Textformat handelt, sind die Daten leicht leserlich, auch für Menschen, und können später erneut in eine Datenbank importiert werden.

Das Dateiformat JSON ist ein simples Format zum Austausch von Daten. Es ist weitverbreitet, unabhängig von Programmiersprachen und viele gängigen Programmiersprachen, wie C, C++ und Java können JSON sowohl lesen, als auch schreiben. Auch viele Datenbanken bieten mittlerweile neben einem XML-Datentyp einen JSON-Datentyp an.

JSON-Dokumente bestehen aus Name/Wert-Paaren, Objekten, die diese Paare beinhalten, sowie Listen von Werten. Da Objekte selber auch als Wert angesehen werden, können diese Strukturen beliebig tief verschachtelt werden. Weitere Werte sind Strings (Zeichenketten), Zahlen (Ketten von Ziffern) und die besonderen Werte *null*, *false* und *true*. Als reines Textformat kann JSON direkt keine binären Daten darstellen, jedoch kann der IM4-Archiver auch binäre Daten archivieren, indem diese Base64 codiert werden. Eine genaue Beschreibung des JSON-Formats lässt sich unter http://json.org/ finden.

Um nicht nur Datenbank-, sondern auch Betriebssystem-unabhängig zu sein, wurde der IM4-Archivier in Java realisiert. Dies erlaubt es, dass Programm auf jedem Betriebssystem aufzurufen, das eine Java Runtime Environment anbietet. Es wird mindestens eine Version ab Java 8 benötigt.

Für die Kommunikation mit Datenbanken wird JDBC (Java Database Connectivity) genutzt. Diese Schnittstelle ist Teil der Java Plattform und ermöglicht einen universellen Zugriff auf verschiedene relationale Datenbanken, sofern es einen JDBC-Treiber für die Datenbank gibt.
JDBC ermöglicht es, normale SQL Anfragen auszuführen und die Rückgabe Werte in Java Objekte umzuformen. Des Weiteren gibt es Methoden zum Abfragen von Metadaten, wie z.B. Tabellennamen, Spaltentypen oder Primary Keys.

Um Datenbank-unabhängig zu sein, richtet sich der IM4-Archiver nach der JDBC-Spezifizierung und bleibt so generisch wie möglich. Jedoch ist die Implementierung der Spezifizierung jedem Treiberhersteller überlassen und diese weichen Teilweise von der Spezifizierung ab, besonders bei ungewöhnlicheren Datentypen. Mehr dazu im Kapitel Export 6 und im Appendix 21 (Bekannte Datentypen 28, Strukturen, Formate und Encodings 32).
Der IM4-Archiver beinhaltet die Treiber für die folgenden Datenbanken:

- Db2
- MSSQL
- PostgreSQL
- Oracle

- SAP HANA

Diese Datenbanken werden im Folgenden als bekannte Datenbanken bezeichnet. Es ist aber auch möglich dem Programmaufruf einen weiteren JDBC-Treiber mitzugeben, um mit einer anderen Datenbank zu kommunizieren.

Neben dem Archivieren von Tabellen und Views im JSON-Format, kann der IM4-Archiver auch die Daten wieder in eine Datenbank importieren. Jedoch handelt es sich hierbei nicht um eine heterogene Systemkopie. Sollten die Tabellen im Zielsystem noch nicht vorhanden sein, wird eine grundlegende Tabelle angelegt, ohne jegliche Indexe, Schlüssel und Ähnlichem. Es wird nur Versucht eine Struktur zu erstellen, die die Daten beinhalten kann. Wenn die Tabellen schon erstellt sind, werden sie vom IM4-Archiver einfach mit INSERT Statements befüllt.

2 Export

Der IM4-Archiver kann Schemas, Tabellen und Views (ohne Definition, werden wie Tabellen behandelt) archivieren. Dabei werden sowohl die Metadaten als auch die Tabelleninhalte in JSON-Dateien auf einem Dateisystem geschrieben. Da die Kommunikation über JDBC stattfindet kann die Datenbank auf einem anderen System laufen, als das von dem der IM4-Archiver aufgerufen wird.

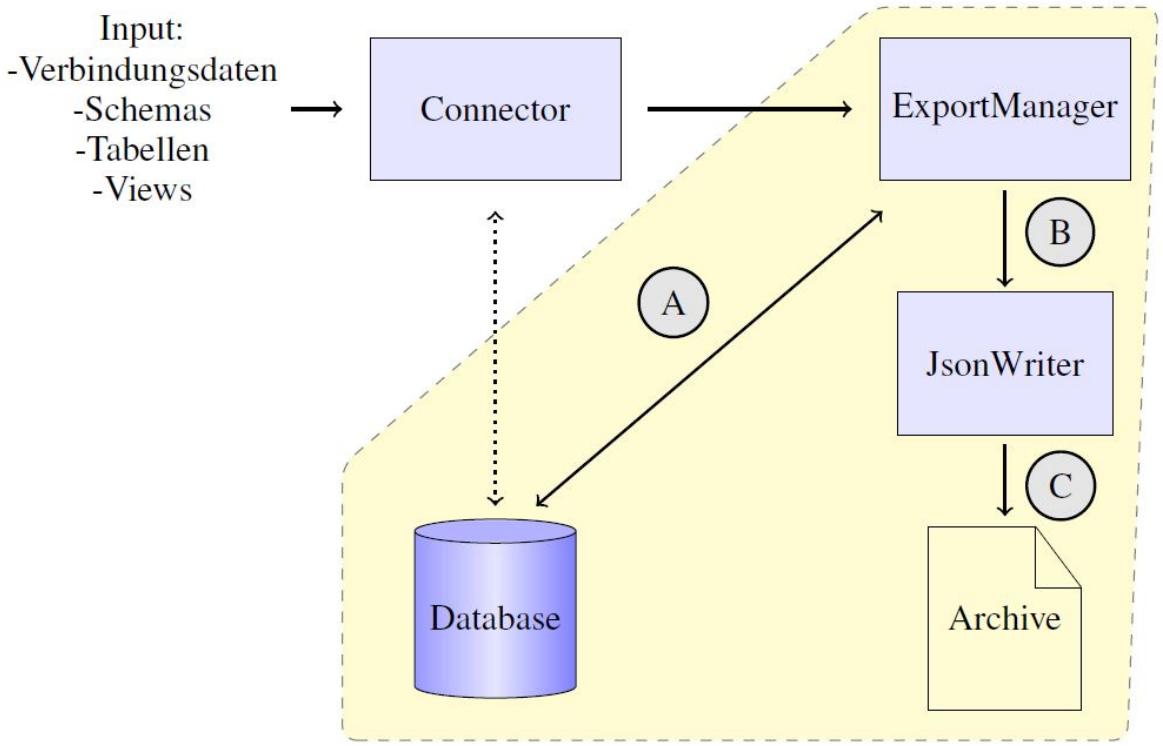

Abbildung 1: Programmabaluf Export

Abbildung 1 1 zeigt den Programmablauf des Exports. Am Anfang wird eine JDBC-Verbindung zur Datenbank hergestellt. Im nächsten Schritt wird überprüft ob die angegebenen Schema, Tabellen und Views tatsächlich in der Datenbank vorhanden sind. Sollte dies der Fall sein, werden die Metadaten der gewünschten Objekte abgefragt und an den Beginn der Archivdatei geschrieben. Danach werden nach und nach die einzelnen Inhalte per SELECT Statements von der Datenbank abgefragt, von der ExportManager Komponente empfangen und Zeile für Zeile von der JsonWriter Komponente in JSON Objekte serialisiert. Am Ende des Exports wird eine zusätzliche Reportdatei erstellt, welche einen Überblick über die archivierten Daten und Archivdateien gibt.

Abbildung 2 skizziert die Verarbeitung der Datenbankantworten. Die Ergebnisse der Anfragen sind sogenannte ResultSets. Diese entsprechen in etwa einem Cursor auf den Ergebnissen und bieten Anwendungen die Möglichkeit die verschiedenen Werte als Java Klassen zu erhalten.
Diese Umwandlung wird von den JDBC-Treibern implementiert. Der IM4-Archiver wählt basierend auf den Metadaten eine geeignete Java Klasse aus und schreibt sie nach und nach in eine Archivdatei. So werden die Zeilen des ResultSets in JSON Objekte umgewandelt.

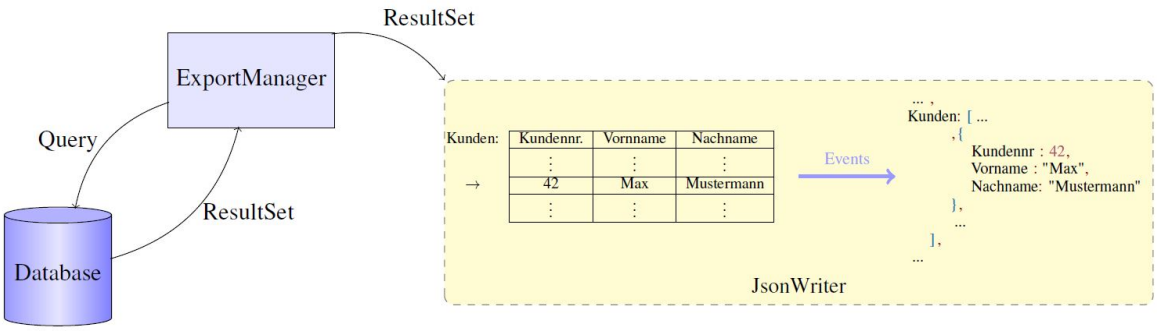

Abbildung 2: Serialisierung der Datenbankobjekte

2.1 Metadaten

Die Abfrage der Metadaten der Tabellen und Views erfolgt nicht direkt über den Katalog einer Datenbank, sondern über die JDBC Methoden zur Abfrage der Metadaten. Somit ist es auch möglich, die Metadaten von unbekannten Datenbanken auszulesen. Die Interpretation der Werte der Metadaten muss demnach der JDBC Spezifizierung folgen. Neben der Zuordnung von Tabellen und Views zu einem Schema beinhalten die Metadaten die folgenden Informationen:

- Eine Beschreibung der Spalten mit dem JDBC-Konstanten für die Datentypen, Spaltennamen, Datentypnamen, Spaltengröße, falls anwendbar Nachkommastellen und ob eine Spalte null sein kann
- Primär- und Fremdschlüssel
- Indexe

Eine genaue Beschreibung der Werte findet man in der JDBC Spezifierung zu den Methoden: *java.sql.DatabaseMetaData.getColumns*, *java.sql.DatabaseMetaData.getPrimaryKeys*, *java.sql.DatabaseMetaData.getImportedKeys* und *java.sql.DatabaseMetaData.getIndexInfo*, sowie in der jeweiligen Dokumentation der JDBC-Treiber.

Da die verschiedenen Treiber teilweise von der Spezifizierung abweichen, ist es notwendig, dass der IM4-Archiver zusätzlich zu den JDBC-Spalten Metadaten eine eigene Abschätzung erstellt. Diese wird beim Export nur genutzt um zu erkennen, wie der Wert einer Spalte gelesen werden soll, aber beim Erstellen der Tabellen beim Import ist die Schätzung notwendig um den richtigen Datentyp auszuwählen. Diese Schätzung nutzt die JDBC-Typ Konstanten, welche auch im Appendix JDBC Generische SQL Datentypen & Konstanten 27 aufgelistet werden. Bei unbekannten Datenbanken ist es nicht möglich so eine Schätzung zu erstellen. Eine Übersicht über die bekannten Datentypen gibt es im Appendix Bekannte Datentypen 28.

2.2 Daten

Das Abfragen der Tabellen- und Viewinhalten erfolgt durch SELECT Statements. In den Statements werden alle Spalten, die in den Metadaten gefundenen wurden, explizit aufgerufen. Sowohl der Schema-, Tabellen- und auch die Spaltennamen sind alle jeweils mit " umgeben. Hierbei ist zu beachten, dass sowohl die Datenbank als auch die JDBC-Treiber die Werte aufbereiten. Die archivierten Werte entsprechen also nicht den tatsächlichen Binärdaten der Datenbank. Diese Binärdaten sind natürlich auch nicht menschenlesbar und in der Regel nicht kompatibel mit anderen Datenbanken.

Um einige Datentypen eindeutig speichern zu können und um bestimmte Fehler in JDBC-Treibern zu umgehen ist es teilweise notwendig die Spalten in andere Datentypen zu konvertieren. So sind zum Beispiel Zeit, Zeitstempel und Datums Werte nicht eindeutig, es sei denn sie sind in einem bekannten Format. Alle archivierten Zeit, Zeitstempel und Datums Werte von bekannten Datenbanken folgen dem ISO 8601 Format:

yyyy-MM-dd“T“hh24:mm:ss.ff..f±HH:mm

IM4-Archiver unterstützt im Moment bis zu 9 fraktionelle Sekundenbruchteile. Alle solche temporalen Datentypen werden datenbankseitig in ein bekanntes Format formatiert und zu einem VARCHAR oder vergleichbaren Datentyp konvertiert. Sollte das Format noch nicht dem ISO 8601 Format entsprechen, versucht der IM4-Archiver den String erneut zu formatieren.

Generell unterstützt der IM4-Archiver die folgenden Datentypen beim Export:

- Jegliche numerischen Datentypen, ausg. die Werte ±Infinity, NaN
- Binäre Datentypen, diese werden Base64 codiert
- Charakter Datentypen, wobei Datentyp mit potentiell großen Werten auch Base64 codiert werden
- Dokumente (XML, JSON), im Falle von XML wird das Dokument binär gespeichert
- Boolean und Bit

Das Abfragen der einzelnen Werten geschieht in bis zu drei Versuchen:

1. Abfrage als Java Klasse basierend auf Abschätzung
2. Abfrage als Java Klasse basierend auf JDBC internes Mapping
3. Abfrage als Java String

Der erste Versuch ist der Normallfall für die bekannten Datenbanken und –typen.
Eine genaue Übersicht über alle dieser Datentypen findet sich im Appendix Bekannte Datentypen 28. In den meisten Fällen kann der IM4-Archiver auch Werte archivieren, für die es keine Schätzung gibt, indem das JDBC interne Mapping genutzt wird und mit generischen Java Klassen wie String, Numerical oder BLOB verglichen wird und als JSON Objekt archiviert wird. Sollte ein Wert auch nicht in diesem Versuch abgedeckt werden, wird der Wert als Java String archiviert.

Im Falle einer unbekannten Datenbank ist die Bedingung für den ersten Versuch nie erfüllt, für die meisten nicht-exotischen Datentypen reichen jedoch die ersten beiden Versuche aus um sinnvolle Werte zu archivieren.

2.3 Dateien & Datenintegrität

Während des Exports können bis zu drei verschiedene Dateiformate erstellt werden: Archiv-, Report- und Prüfsummen-Dateien. Die Struktur der Dateien ist im Appendix Strukturen, Formate und Encodings 32 beschrieben.
Die Archivdateien, welche die Metadaten und die Records der Tabellen und Views enthalten, sind im JSON Format geschrieben und UTF-8 codiert. Zusätzlich können diese Archivdateien auch noch GZIP komprimiert. Dieses Kompressionsverfahren beinhaltet auch CRC32-Prüfsummen. Aber auch für nicht-komprimierte Archivdateien werden diese Prüfsummen berechnet.
Des weiteren können Archivdateien auch direkt mit dem AES oder ABE Verfahren aus dem IM4Crypto Programm verschlüsselt werden.

Jede Archivdatei beginnt mit Metadaten, diese sind immer mind. ausreichend um den eigenen Inhalt zu beschreiben und importieren zu können. Genauer gesagt beinhaltet eine Archivdatei immer alle Metadaten, der noch nicht komplett archivierten Tabellen und Views. Wenn ein Dateigrößelimit angegeben wird, kann es sein dass der IM4-Archiver mehrere Archivdateien erstellt. Falls ein Limit existiert, wird vor dem Schreiben einer Zeile immer eine grobe Schätzung gemacht, ob die neue Zeile das Limit überschreiten würde. Wenn dem der Fall ist, dann wird die aktuelle Datei geschlossen, eine neue Datei erstellt, die Metadaten aller noch nicht exportierten Objekte geschrieben, gefolgt von der Zeile welche das Limit überschritten hat.

Am Ende eines Exports wird eine Reportdatei erstellt. Genau wie die Archivdatei, ist die Reportdatei eine UTF-8 codierte JSON Datei, mit einer JSON Schema Beschreibung im Appendix. Sie beinhalten eine Übersicht über das Ergebnis eines Exportaufrufes, insbesondere über die erstellten Archivdateien, deren Inhalt (Tabellen & Views, Anzahl an Records) und Prüfsummen, sowie eine Angabe der totalen Anzahl der archivierten Sätze pro Tabelle und View.
Des Weiteren stehen Datenbankname, -produkt und –version, UserId, Start- und Endzeitpunkt (Java Systemzeit, Millisekunden seit Epoch) und optionalen Remark Feldern im Report. Letztere sind bis zu drei JSON Name/Werte-Paare „REMARK“, „REMFILE“ und „REMTYPE“. Das erste und dritte Paar können einen String beinhalten, das letzte Paar beinhaltet eine binäre Datei, Base64 codiert. Die Intention ist es hier die Begründung für die Archivierung zu beschreiben. Das erste Paar bietet die Möglichkeit für eine einfache Beschreibung als String. Falls aber eine längere Begründung hinterlegt werden soll, welche z.B. in einem PDF Dokument beschrieben ist, ist das Feld „REMTYPE“ dazu gedacht, den Dateityp zu beschreiben und „REMFILE“ um die Datei im Report zu speichern.

Der letzte und optionale Dateityp sind die Prüfsummen-Dateien, diese beinhalten Prüfsummen, welche zum Überprüfen der Datenintegrität auf Zeilenebene genutzt werden können.
Falls erwünscht, können die Java Klassen beim Export pro Zeile eine CRC32-Prüfsumme erstellt werden. D.h. die Werte einer Zeile werden parallel zum Schreiben der Archivdatei auch als RC32 Prüfsumme in die Prüfsummen Datei geschrieben. Diese Vorgehen ist in Abbildung 3 beschrieben

Später ist es dann möglich, die Werte die im Archiv stehen, erneut zu hashen und mit dem Wert in der Prüfsummen zu vergleichen. Dadurch kann man die Integrität der einzelnen Zeilen überprüfen, ähnlich wie die CRC32-Prüfsummen der Dateien. Aber auch vor dem Einfügen einer Zeile kann dieser Vergleich genutzt werden. Die Prüfsummen werden mit Werten berechnet die vom JDBC Treiber abgefragt wurden, beim Import

werden die Werte überprüft bevor sie dem JDBC Treiber übergeben werden. Sowohl der Treiber als auch die Datenbank können natürlich die Werte noch ändern.

Im Gegensatz zu den Archiv- und Reportdateien sind die Prüfsummen Dateien nicht im JSON Format. Es handelt sich um reine Textdateien in der Zeile für Zeile je ein CRC32-Wert geschrieben wird. Da die Ergebnisse von Hash-Funktionen binär sind, muss jeder CRC32-Wert Base64 codiert werden. Neben den Hash-Werten gibt es in den Prüfsummen Dateien auch Kommentare, diese sind mit einem ‚#' an der ersten Stelle einer Zeile markiert. Die Kommentare beschreiben zu welcher Tabelle in welcher Datei die Prüfsummen gehören, sowie wie die aktuelle Zeilennummer.

Es ist wichtig zu erwähnen das die genutzten Verfahren keine Geheimnisse oder ähnliches beinhalten. Die Daten sind nicht gegen mutwillige Änderungen geschützt.

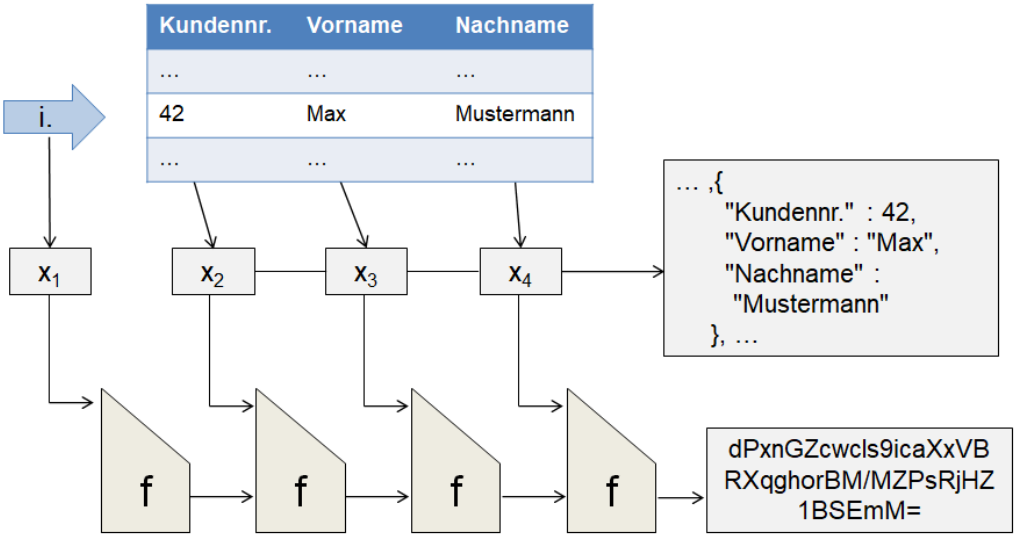

Abbildung 3: Berechnung der Hashsummen

Alle Dateien die während eines Exportlaufs erzeugt werden folgen dem gleichen Namensschema. Dem Programm wird ein Prefix mitgegeben, im folgenden als X bezeichnet, und dann je nach Dateityp und Exporteinstellungen wird der restliche Dateiname erzeugt. Die Archivdateien haben neben der .json Dateiendung zusätzlich noch die Endungen .gz, .aes und .cpabe, je nachdem ob sie komprimiert und/oder verschlüsselt sind. Die erste Archivdatei heißt immer X.json, falls es ein Dateilimit gibt und dies erreicht wurde, heißen alle weiteren Dateien X.YYYY.json(.gz), wobei YYYY ein Zähler ist, der pro Datei inkrementiert wird und bei 0000 startet. Die erste Prüfsummen-Datei heißt X.check und bezieht sicht auf die X.json Datei. Für jede weitere Archivdatei wird auch eine neue Prüfsummen-Datei erstellt mit dem Namensschema X.YYYY.check, dabei sind die Zähler jeweils identisch. Die Reportdatei hat den Namen X.Report.json.

2.4 Export Beispiel

Das folgende Beispiel zeigt den Programmaufruf für einen Exportbefehl der bestimmte Datenbankobjekte aus der Datebank, dessen Aufbau in Abbildung 4 beschrieben ist, entladen soll.. Dabei sollen aus der Datenbank Bsp1, die Tabelle Test1 und alle Tabellen und Views in Schema2 archiviert werden. Für das Beispiel wird angenommen dass die Datenbank von einem dem IM4-Archiver bekannten DBMS verwaltet wird, welches mit dem Platzhalter X beschrieben wird. Zusätzlich soll in der Reportdatei dokumentiert werden, dass dies nur ein Test sei und die erzeugten Archivdateien sollen komprimiert und mit Pretty-Printing (Zeilenumbrücken und -einrückungen) erzeugt werden. Die Dateien, die das Programm erzeugt, sollen in das Verzeichnis C:\Documents\archiv geschrieben werden und mit dem Prefix Beispiel anfangen.

Eine genaue Übersicht über die Programmparameter und den Programmaufruf gibt es im Appendix Programmaufruf und Parameter 22.

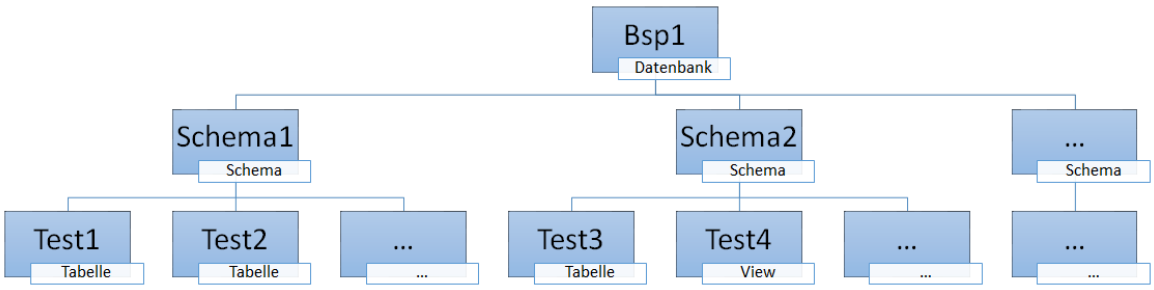

Abbildung 4: Beispiel Datenbankstruktur

Ein Programmaufruf der diesen Anforderungen entspricht wäre:

export -a 127.0.0.1:1234 -n Bsp1 -u test -p test -d C:\Documents\archiv Beispiel -m X -t Schema1.Test1 -s Schema2 -y -z -r "Dies ist ein Test"

Der Aufruf beginnt mit der Angabe des Befehls, der ausgeführt werden soll. In diesem Fall soll exportiert werden, also *export*.
Die im folgenden aufgelisteten Parameter sind für das I/O des Programmes notwendig:

- *-a 127.0.0.1:1234:* Host:Port der Quelldatenbank
- *-n Bsp1:* Bezeichner/Name der Quelldatenbank
- *-u test -p test:* User und Passwort mit dessen Berechtigung der IM4-Archiver sich verbinden soll
- *-d C:\Documents\archiv Beispiel:* Verzeichnis und Dateinameprefix
- *-m X:* Name des DBMS

Als nächstes folgen die export-spezifischen Parameter, jedoch ist die Reihenfolge der Parameter zufällig und muss nicht eingehalten werden.

- *-t Schema1.Test1*: Fügt die Tabelle Schema1.Test1 in die Menge der Objekte, die exportiert werden sollen
- *-s Schema2:* Fügt alle Tabellen und Views des Schema2 in die Menge der Objekte, die exportiert werden soll
- *-y*: Aktiviert Pretty-Printing
- *-z:* Aktiviert Kompression
- *-r "Dies ist ein Test":* Erzeugt in der Report-Datei das Name/Wert Paar: REMARK: "Dies ist ein Test"

Der IM4-Archiver unterscheidet nicht zwischen Tabellen und View, d.h. die Parameter -t, -s und -l funktionieren mit Tabellen und Views. Sollte man gezielt noch die Tabelle Test2 archivieren wollen, kann man den -t Parameter erweitern zu *-t Schema1.Test1 Schema1.Test2.* Objekte in anderen Schema können zu dieser Liste hinzugefügt werden, wie z.B. Schema2.Test4. Wenn ein Objekt mehrmals aufgelistet wird, wird es nur einmal exportiert. Auch der -s Parameter verhält sich wie eine List, man kann mit -s Schema2 Schema1 die kompletten Inhalte der beiden Schema archivieren.

Wenn es das Verzeichnis C:\Documents gibt, dann erzeugt der IM4-Archiver entweder das Verzeichnis \archiv oder nutzt es, falls das Verzeichnis schon existiert. In diesem Verzeichnis erstellt das Programm die Dateien Beispiel.json.gz und Beispiel.Report.json. Sollte es schon Dateien mit dem Prefix Beispiel im Verzeichnis geben, bricht das Programm ab.

3 Import

Mit dem Import Befehl können die Daten und die Tabellen und Views der Archivdateien wieder in eine Datenbank importiert werden. Der Programmablauf, gezeigt in Abbildung 5, ist ähnlich wie der Export. Am Anfang wird wieder die JDBC Verbindung hergestellt und überprüft ob es im angegebenen Verzeichnis überhaupt ein Archiv mit dem gegebenen Namen gibt. Ist dies nicht der Fall bricht das Programm ab. Danach werden die Metadaten der ersten Archivdatei gelesen. Diese Metadaten sind später notwendig sowohl zum Lesen der Archivdatei, aber auch zum Erstellen der Datenbankstrukturen, falls diese noch nicht in der Datenbank vorhanden sind.

Im nächsten Schritt wird überprüft welche Strukturen (Schema und Tabellen) es schon gibt und welche noch erstellt werden müssen. Letztere werden dann vom IM4-Archiver erstellt. Danach wird das Archiv gelesen und die Zeilen mit INSERT Statements in die Datenbank importiert.

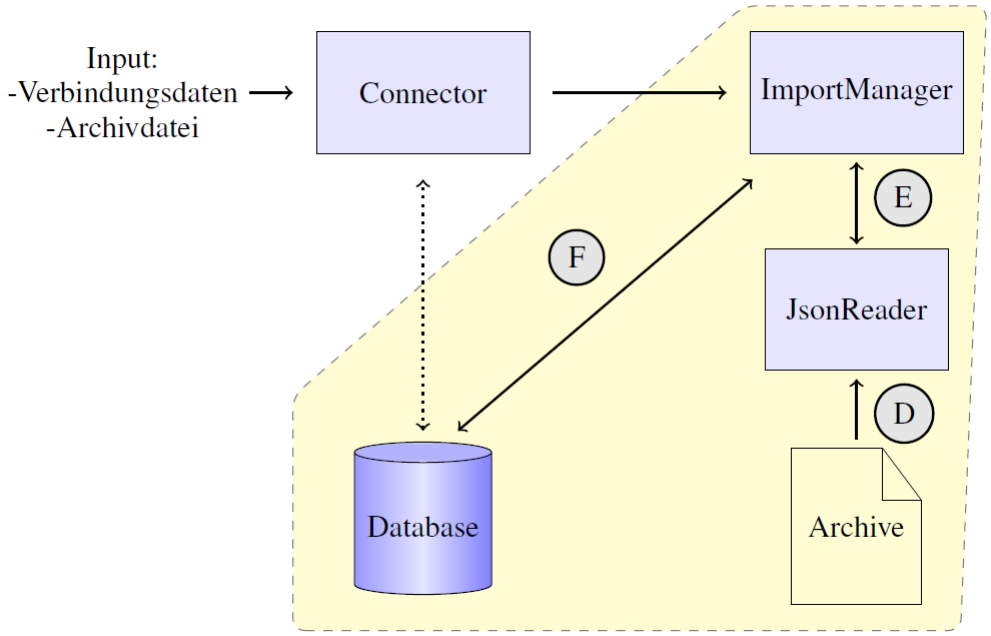

Abbildung 5: Programmablauf Import

Natürlich muss nicht immer das ganze Archiv importiert werden, man kann auf Basis der Schema- und Tabellennamen filtern, indem man unter anderem explizit Schema und Tabellen ignoriert . Es ist auch möglich die genutzten Namen beim Import zu ändern. Eine genau Übersicht über die Parameter des Imports befindet sich im Appendix Programmaufruf und Parameter 22. Ähnlich wie Export sind auch beim Import alle Schema-, Tabellen- und Spaltennamen mit " umgeben.

3.1 Datenbankstrukturen

Wie Anfangs schon erwähnt erstellt der IM4-Archiver Datenbankstrukturen, falls diese noch nicht vorhanden sind. Jedoch handelt es sich hierbei immer nur um sehr einfache Tabellen. Es werden keinerlei Eigenschaften wie Primär-, Fremdschlüssel, Indexe, Einschränkungen auf die Spaltenwerte oder Ähnliches angelegt. Auch die Views, die exportiert wurden, werden beim Import genau wie eine Tabelle behandelt. Es ist nicht das Ziel der vom IM4-Archiver erstellten Tabellen besonders performant, produktionsfähig oder eine exakte Kopie der exportierten Tabelle zu sein. Sondern das Ziel ist es die Daten des Archivs wieder darstellen zu können. Falls die Tabellen schon existieren, dann wird das Erstellen übersprungen und der IM4-Archiver versucht die Daten zu importieren, basierend auf der Tabellenstruktur, die im Archiv steht.

Falls die Tabellen noch nicht existieren, erstellt der IM4-Archiver eine einfache Tabelle. Da verschiedene Datenbanken jedoch verschiedene Datentypen unterstützen, bzw. verschiedene Namen für Typen haben, die sich identisch verhalten, ist es also wichtig Anhand der JDBC und Archiv Metadaten den richtigen Datentypnamen der Zieldatenbank auszuwählen. Auch die richtigen Einschränkungen für die Datentypen sind wichtig. Mit Einschränkungen sind hier gemeint, z.B. die Präzision und Skala bei Decimal Datentypen oder die Anzahl an Sekundenbruchteilen bei Timestamps. Die Werte für die Einschränkung stehen in jeder Spalte in den Name/Wert-Paaren COLUMN_SIZE und DECIMAL_DIGITS. Zum Beispiel wird bei einem Varchar Datentyp die COLUMN_SIZE als Anzahl der Charakter interpretiert.

Die Auswahl des Datentypnamens und der Einschränkung erfolgt nach den folgenden drei Fällen:

1. Quelldatenbank = Zieldatenbank:
    i. Datentypname wird übernommen
    ii. Einschränkung basierend auf SQL Namen für den generischen SQL Typ der Schätzung, falls vorhanden
    iii. Einschränkung basierend auf dem generischen SQL Typ des JDBC Treibers beim Export
2. Quelldatenbank ≠ Zieldatenbank & es gibt eine Schätzung:
    i. Datentypname basierend auf dem Namen für den generischen SQL Typ der Schätzung
    ii. Gleiches gilt für die Einschränkung
3. Quelldatenbank ≠ Zieldatenbank & es gibt keine Schätzung:
    i. Datentypname und Einschränkung basierend auf dem generischen SQL Typ des JDBC Treibers beim Export

Der Vergleich der Datenbanken basiert auf der Liste der bekannten Datenbanken, sowie dem Sonderfall OTHER, welcher stellvertretend für alle unbekannten Datenbanken steht. D. h. Export von einer unbekannten Datenbank X und Import in eine andere unbekannte Datenbank Y trifft den Fall 1. Da unbekannte Datenbanken auch unbekannte Namen für die generischen SQL Typen haben wird in so einem Fall auch einfach der Datentypname des JDBC Treibers beim Export übernommen.

3.2 Daten

Sobald die Tabellen erstellt sind, werden sie mit INSERT Statements in JDBC-Batch Verarbeitung befüllt. Dabei wird die JSON Datei Stück für Stück gelesen und die Werte, den Metadaten entsprechend, in Java Objekte umgewandelt. Diese werden dem JDBC-Treiber übergeben und als ein INSERT Statement an die Datenbank geschickt.

Sollte es beim Importieren einer Tabelle zu einer SQLException kommen, wird der Import der Tabelle abgebrochen und es wird versucht die restlichen Tabellen zu importieren.

Genau wie beim Export gibt es auch hier eine Aufbereitung durch den JDBC-Treiber und der Datenbank.

3.3 Import Beispiel

In diesem Beispiel wird das Archiv, welches im Export Beispiel 13 erzeugt wurde, in die Datenbank Bsp2 importiert. Der Aufbau der Datenbank vor und nach dem Import ist beschrieben in Abbildung 6 und 7. Mit den archivierten Daten gibt es jedoch Namenskonflikte. Beide Datenbanken haben Schema mit den Namen Schema1 und Schema2 und Bsp2 hat eine Tabelle mit dem Namen Schema2.Test4. Da das Importieren mittels INSERT Statements erfolgt, würde das Programm ohne Auflösung dieses Konfliktes versuchen , den Archivinhalt von Bsp1.Schema2.Test4 in die Tabelle Bsp2.Schema2.Test4 einzufügen. Für dieses Beispiel soll die archivierte Tabelle Test4 in Test4Copy umbenannt werden und alle Tabellen des archivierten Schema1 sollen in ein neues Schema Schema1Copy importiert werden.

Eine genaue Übersicht über die Programmparameter und den Programmaufruf gibt es im Appendix Programmaufruf und Parameter 22.

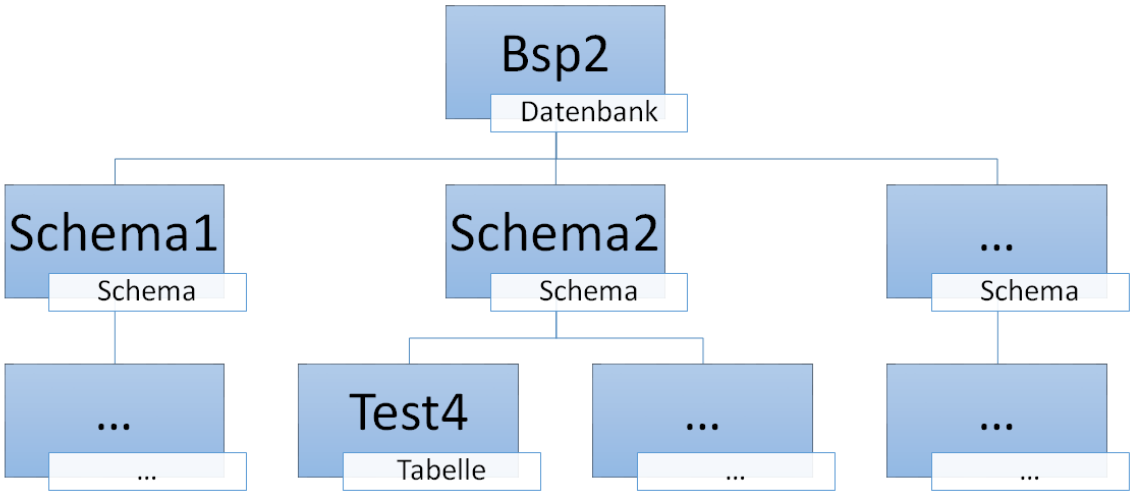

Abbildung 6: Beispiel Datenbankstruktur vorher

Ein Programmaufruf, der diesen Anforderungen entspricht, wäre:

import -a 127.0.0.1:1234 -n Bsp2 -u test -p test -d C:\Documents\archiv Beispiel -m X --cs Schema1 Schema1Copy --is Schema2 --ct Schema2.Test4 Schema2.Test4Copy

Eine Erklärung für die I/O Parameter findet sich im Export Beispiel 13 und dem Appendix.

- *--cs Schema1 Schema1Copy:* Alle Objekte im Schema1 werden als Objekt von Schema1Copy importiert
- *--is Schema2:* Alle Objekte im Schema2 werden ignoriert, bis auf Objekte die mit anderen Parametern verändert werden
- *--ct Schema2.Test4 Schema2.Test4Copy:* Schema2.Test4 wird als Schema2.Test4Copy importiert

Ähnlich wie beim Export handelt es sich bei diesen Parametern und ihren Werten wieder um Listen, d.h. neue Einträge können einfach angehängt werden. Jedoch gibt es bei einigen Import Parametern zusammenhänge zwischen den Einträgen, so muss z.B. bei den --ct Parameter immer der alte Name gefolgt vom neuen, angegeben werden.

In diesem Beispiel werden nur zwei Objekte aus dem gesamten Archiv importiert. Da es noch kein Schema1Copy.Test1 gibt, erzeugt der IM4-Archiver das Schema und die Tabelle. Ähnliches gilt für Schema2.Testcopy, jedoch ist hier zu beachten, dass es das Schema2 schon gibt und dass Schema2.Test4Copy als eine Tabelle erzeugt wird. Auch beim Import werden Tabellen und View gleichbehandelt vom Programm. Nachdem die neuen Objekte erzeugt wurden, in der Abbildung mit '*' markiert, werden sie mit INSERT-Statements befüllt.

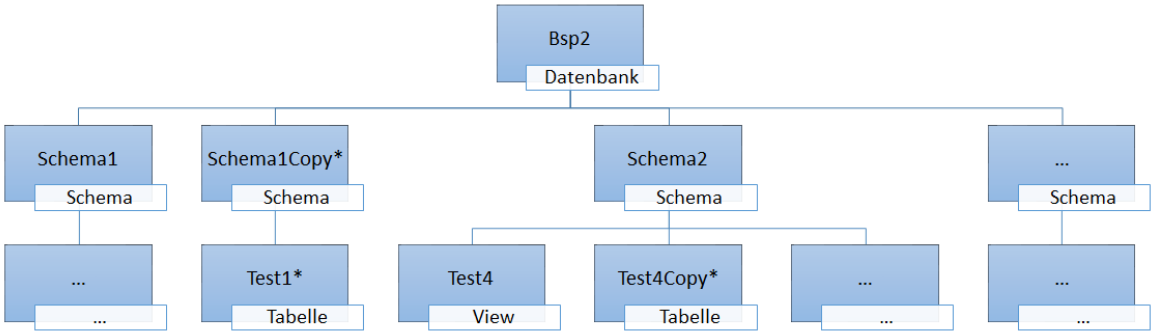

Abbildung 7: Beispiel Datenbankstruktur nachher

4 Appendix

4.1 Programmaufruf und Parameter

Die Argumente des IM4-Archivers haben die Form:

BEFEHL PARAM1 [Wert1.1 Wert1.2 …] PARAM2 …

Wobei es die folgenden Befehle gibt: „export“, „import“, „check“, „help“, "generate" und "version". Das Verhalten der ersten beiden Befehle wurde im Detail in den Kapiteln Export 6 und Import 15 beschrieben. Der Befehl „check“ überprüft die berechneten Prüfsummen und der Befehl "version" gibt die Version des Programms, sowie die Version aller dem Programm zur Laufzeit bekannten JDBC Treiber, aus. Der "generate" Befehl schreibt die DDL Befehle, die ein Import erzeugen würde, in eine Textdatei raus.

Der Befehl „help“ gibt eine genaue Übersicht über die Befehle, deren Parameter und ihr Verhalten aus. Wird der Hilfebefehl mit keinen weiteren Argumenten aufgerufen, werden Informationen zu allen Befehlen ausgegeben. Alternativ kann man mit den Argumenten "help BEFEHL" gezielt die Parameterliste eines bestimmten Befehls abfragen. Der "help" Befehl kann auch genutzt werden um eine JSON-Schema Beschreibung der Archiv- und Reportdatei auf der Konsole auszugeben. Hierzu wird der Wert "schema" übergeben.

Die folgenden Tabellen bieten eine Übersicht über die Parameter der Befehle. Da sowohl beim Export als auch beim Import eine Datenbankverbindung, sowie ein Verzeichnispfad und Archivnamensschema gegeben sein muss, teilen sich diese Befehle einige Parameter, im Folgenden als allgemeine Parameter beschrieben. Der Befehl "version" hat keine Argumente und der Befehl "help" entweder keins oder den Namen eines anderen Befehls oder dem Schlüsselwort "schema".

Die erste Spalte der Tabellen listet die verschiedenen Namen der Parameter und die Stelligkeit der Parameter. Die Stelligkeit ist die in Klammern stehende Zahl, welche beschreibt wie viele Werte ein Parameter braucht. So bedeutet (0) dass ein Parameter keine Werte benötigt, (1) dass er einen Wert benötigt und (n) dass er ein beliebige Anzahl an Werten benötigt. Eine Stelligkeit von (2*n) bedeutet dass eine gerade Anzahl an Werten benötigt wird. Die Stelligkeit (x-y) bedeutet das es zwischen x und y Werte geben kann.

Die zweite Spalte gibt eine Beschreibung über die Wirkung der Parameter als auch deren Werte.

Alle Shell Argumente sind leerzeichengetrennt, sollte ein Leerzeichen teil eines Wertes sein, wie z.B. in einem Verzeichnispfad oder in einem Satz, muss dieser Wert je nach Shell richtig quotiert werden. Parameter mit variabler Stelligkeit können auch mehrmals erwähnt werden, es macht keinen Unterschied ob man *-t Table1 Table2 -u ...* oder *-t Table1 -u ... -t Table2* nutzt. Werte werden einem Parameter zugeordnet bis ein Wert gefunden wird, der selber ein neuer Parameter ist. Es ist auch möglich Parameter in einer Datei zu speichern und bei einem Aufruf den Dateipfad anzugeben. Dafür muss ein Argument mit "!", "@" oder "<" beginnen, gefolgt von dem Dateipfad. Die Argumente in diesen Datei sind mit Zeilenumbrüchen getrennt.

Parameter	Beschreibung
1. Allgemeine Parameter	
-a, --address (1)	IP und Port im Format: A.B.C.D:E

-n, --name (1)	Name der Datenbank
-u, --user (1)	Benutzername
-p, --password (0-1)	Passwort • Kein Wert: Passwort wird auf der Konsole abgefragt • Ein Wert: Wert wird als Passwort genutzt
-d, --directory (1-2)	Namenschema, insbesondere Verzeichnispfad • Ein Wert: Verzeichnispfad, Name des letzten Elements des Pfades wird als Namenschema genutzt. Beim Import kann hier auch direkt eine Datei angegeben werden • Zwei Werte: Erster Wert ist der Verzeichnispfad, zweiter Wert ist das Namenschema des Archives • Ein Verzeichnis kann mehrere Archive enthalten, solange diese unterschiedliche Namenschema haben
-m, --dbms (1)	DBMS • Mögliche Werte: [DB2, DB2LUW, MSSQL, POSTGRESQL, HANA, ORACLE, OTHER] • Falls der OTHER Typ ausgewählt wird, muss der –otherURL Parameter angegben werden
--other, --otherURL (1)	JDBC-URL für eine dem Programm unbekannten Datenbank • Übliche Form: jdbc:Product://Host:Port/Database • User und Passwort können weiterhin über –user und –password angegeben warden • JDBC-Treiber muss sich im Klassenpfad befinden und für das automatische Registrieren JDBC4 konform sein
--jdbcProperty (n)	Eine oder mehrere JDBC-Properties die zur Verbindungsherstellung genutzt werden sollen • Properties müssen die Form X=Y haben • Properties werden während des Verbindungsaufbaus geloggt • Properties können das erwartete Verhalten des Programms stören

Tab. 1

2. export Parameter	
-s, --schemas (n)	Liste von Schemanamen • Exportiert alle Tabellen und Views in den angegeben Schemas • Schemaname kann ein voller Name sein: „SCHEMA“ • Schemaname kann ein SQL-„LIKE“ Muster sein: „SCHE%“, jedoch wird nur der erste Eintrag des Musters archiviert (siehe –tablePattern)

-t, --tables (n)	Liste von Tabellennamen • Exportiert die angegebenen Tabellen und Views • Tabellenname kann ein voller Name sein: „SCHEMA.TABLE“ • Tabellenname kann ein SQL-„LIKE“ Muster sein, jedoch wird nur der erste Eintrag archiviert
-l, --likePattern (n)	Liste von SQL-„LIKE“ Mustern • Exportierte alle Tabellen und Views, die dem Muster entsprechen (im Gegensatz zu –schema und –table)
--gf, --globalFetch (1)	Gewünschte Anzahl an Reihen eines ResultSet • Setzt den JDBC fetchSize Hint für alle ResultSets • Performance Tuning: Netzwerkanfragen gegen Speicherbedarf • Default: Abhängig vom JDBC Treiber
--tf, --tableFetch (2*n)	Liste der gewünschte Anzahl an Reihen eines ResultSet für eine spez. Tabelle • Setzt den JDBC fetchSize hint für die angegebenen ResultSets • Besteht aus Paaren „SCHEMA.TABLE FETCHSIZE“ • Wie –globalFetch jedoch nur für SELECT Statements auf bestimmte Tabellen, überschreibt Wert von --globalFetch
-i, --transactionIsolationLevel, --isolation (1)	Transaktion Isolations Level • Setzt den JDBC Hint für das Transaction Isolation Level • Mögliche Werte: [READ_UNCOMMITTED(1), READ_COMMITTED(2), REPEATABLE_READ(4), SERIALIZABLE(8)] • Entweder der volle Begriff oder die Zahl kann genutzt werden • Default: READ_COMMITTED
-y, --pretty (0)	Aktiviert Pretty Printing • Fügt Zeilenumbrüche und Einrückungen dazu
-z, --compression (0)	Aktiviert die GZIP-Kompression
--encrypt (2-3)	Aktiviert die Verschlüsselung der Archivdateien • Erster Wert wählt die Verschlüsselungsart: AES oder ABE • Zweiter Wert ist der AES Schlüssel oder die ABE Public Parameter • Dritter Wert ist nur notwendig für ABE und beschreibt die Access Policy gegeben in Postfixnotation (s. lm4Crypt)
--buffer, --compressionBuffer (1)	Größe des Kompressionspuffers
--fileSize, --limit (1)	Dateigrößenlimit
-h, --hash (0)	AKtiviert die Berechnung der SHA-256 Hashes

-r, --remark (1-3)	Bemerkung für den Report • Erster Wert: String der in den Report geschrieben wird • Zweiter Wert: Verzeichnispfad zu einer Datei, die BASE64 codiert in den Report geschrieben wird • Dritter Wert: String, der genutzt werden kann um den Dateityp zu beschreiben

Tab. 2

3. import Parameter	
-b, --batch (1)	Anzahl der Inserts pro Batch • Default: 250
--cs, --changeSchema (2*n)	Liste der Änderungen an Schemanamen • Besteht aus Paaren: OLD_SCHEMA NEW_SCHEMA • OLD_SCHEMA ist der Name eines Schema im Archiv (Case-Insensitive) • Beim Import wird NEW_SCHEMA verwendet • Überschrieben von -changeTable
--ct, --changeTable (2*n)	Liste der Änderungen an Tabellennamen • Besteht aus Paaren: OLD_SCHEMA.OLD_TABLE NEW_SCHEMA.NEW_TABLE • OLD_SCHEMA.OLD_TABLE ist der Names einer Tabelle oder View im Archiv • Beim Import wird NEW_SCHEMA.NEW_TABLE genutzt
--is, --ignoreSchema (n)	Liste der Schema, deren Inhalte nicht importiert werden sollen • Besteht aus Schemanamen SCHEMA aus dem Archiv • Überschrieben von --changeSchema, --changeTable und --onlyTable
--it, --ignoreTable(n)	Liste der Tabellen, die nicht importiert werden • Besteht aus Tabellennamen SCHEMA.TABLE aus dem Archiv • Überschrieben von --changeSchema, --changeTable und --onlyTable
--ot, --onlyTable (n)	Liste der Tabellen, die als einzige importiert werden • Besteht aus Tabellennamen SCHEMA.TABLE aus dem Archiv • Nur diese Tabellen werden importiert, alle anderen werden ignoriert • Der Name dieser Tabellen kann noch mit -changeTable oder -changeSchema geändert werden

--onlyDDL (0)	Deaktiviert INSERT-Statements
--decrypt (2)	Aktiviert die Entschlüsselung der Archivdateien • Erster Wert wählt die Verschlüsselungsart: AES oder ABE • Zweiter Wert ist die Schlüsseldatei
-h, --hash (0)	Aktiviert das Vergleichen der SHA-256 Hashes
--noChecksums (0)	Deaktiviert das Überprüfen der CRC-32 Prüfsummen der Archivdateien

Tab. 3

Der Aufruf des Programms in der Kommandozeile erfolgt entweder direkt über die ausführbare Jar-Datei mit:

java –jar infodesign.im4archiver.jar BEFEHL …

oder durch Einfügen der Jar Datei im Java Klassenpfad und explizites Auswählen der Main-Klasse:

java –cp …;infodesign.im4archiver.jar infodesign.im4archiver.IM4Archiver BEFEHL …

Letztere Variante ist notwendig wenn z.B. andere JDBC-Treiber genutzt werden sollen. Diese müssen dann mit im Klassenpfad stehen

4.2 JDBC Generische SQL Datentypen & Konstanten

Generischer Datentyp	Konstante
BIT	-7
TINYINT	-6
SMALLINT	5
INTEGER	4
BIGINT	-5
FLOAT	6
REAL	7
DOUBLE	8
NUMERIC	2
DECIMAL	3
CHAR	1
VARCHAR	12
LONGVARCHAR	-1
DATE	91
TIME	92
TIMESTAMP	93
TIMESTAMP_WITH_TIMEZONE	2014
BINARY	-2
VARBINARY	-3
BLOB	2004
CLOB	2005
BOOLEAN	16
ROWID	-8
NCHAR	-15
NVARCHAR	-9
SQLXML	2009
OTHER	1111

4.3 Bekannte Datentypen

Datenbank	Datentyp	Bemerkung
Db2	BIGINT	
-	INTEGER	
-	SMALLINT	
-	DECIMAL	
-	NUMERIC	
-	DOUBLE	
-	REAL	
-	DATE	*CHAR(col, ISO)*
-	TIME	*CHAR(col, JIS)*
-	TIMESTAMP	*CHAR(col),* nur bis zu 9 frakt. Sekunden unterstützt, ISO Formatierung durch den IM4Archiver
-	TIMESTAMP WITH TIME ZONE	
-	CHAR	
-	VARCHAR	
-	CLOB	
-	BINARY	
-	VARBINARY	
-	CHAR FOR BIT DATA	BINARY
-	VARCHAR FOR BIT DATA	VARBINARY
-	BLOB	
-	DECFLOAT	Abschätzung als FLOAT, andere Datenbanken kennen wahr. kein Decimal Floating Point +/- Infinity, NaN nicht unterstützt
-	ROWID	Byte-Wert der ROWID
-	XML	
Oracle	NUMBER	NUMBER ohne Einschränkung wird als FLOAT abgeschätzt. INTEGER für Ganzzahlige NUMBER Spalten und ansonsten DECIMAL.
-	BINARY_FLOAT	+/- Infinity, NaN nicht unterstützt
-	BINARY_DOUBLE	
-	DATE	Abgeschätzt als Timestamp *TO_CHAR(col, , 'YYYY-MM-DD"T"HH24:MI:SS')*
-	TIMESTAMP	*TO_CHAR(col, , 'YYYY-MM-DD"T"HH24:MI:SS.FF9')*

-	TIMESTAMP WITH TIME ZONE	*TO_CHAR(col, , 'YYYY-MM-DD"T"HH24:MI:SS.FF9TZH:TZM')*
-	TIMESTAMP WITH LOCAL TIME ZONE	Abhängig von der Zeitzone der JVM, Cast so wie bei Oracle Timestamp
-	CHAR	
-	VARCHAR2	
-	CLOB	
-	RAW	
-	BLOB	
-	XMLTYPE	
MSSQL	BIGINT	
-	INT	
-	SMALLINT	
-	TINYINT	
-	BIT	
-	DECIMAL, NUMERIC	
-	FLOAT	
-	REAL	
-	DATE	*CONVERT(VARCHAR(40), col, 126)*
-	DATETIME2	
-	DATETIME	
-	SMALLDATETIME	
-	DATETIMEOFFSET	
-	TIME	
-	CHAR	
-	VARCHAR	
-	NCHAR	
-	NVARCHAR	
-	TEXT	
-	NTEXT	
-	BINARY	
-	VARBINARY	
-	IMAGE	BLOB
-	MONEY	DECIMAL
-	SMALLMONEY	DECIMAL

-	TIMESTAMP	BINARY
-	UNIQUEIDENTIFIER	CHAR
-	SYSNAME	NCHAR
-	_ IDENTITY Columns	Schätzung als grundlegender Datentyp, keine Unterscheidung möglich
-	XML	
PostgreSQL	BIGINT	
-	INTEGER	
-	SMALLINT	
-	DECIMAL, NUMERIC	
-	DOUBLE PRECISION	
-	REAL	
-	BIGSERIAL	
-	SERIAL	
-	SMALLSERIAL	Ununterscheidbar zu SMALLINT
-	DATE	*TO_CHAR(col, 'YYYY-MM-DD')*
-	TIME	*TO_CHAR(col, 'HH24:MI:SS')*
-	TIMETZ	*TO_CHAR((col at time zone 'UTC')::time, 'HH24:MI:SS')*
-	TIMESTAMP	*TO_CHAR(col , 'YYYY-MM-DD"T"HH24:MI:SS.US')*
-	TIMESTAMP WITH TIME ZONE	*TO_CHAR(col at time zone 'UTC' , 'YYYY-MM-DD"T"HH24:MI:SS.US"+00:00"')*
-	CHAR	
-	VARCHAR	Abschätzung als CLOB falls unbegrenzt
-	TEXT	Abschätzung als CLOB
-	BOOL	
-	BIT	*col::varchar*
-	VARBIT	*col::varchar*, Abschätzung als CLOB falls unbegrenzt, ansonsten Varchar
-	BYTEA	Abschätzung als BLOB
-	JSON	Abschätzung als CLOB
-	JSONB	Abschätzung als CLOB
-	XML	
-	Geometrische: (LINE, LSEG, POINT, BOX, PATH, POLYGON, CIRCLE)	Abschätzung als CLOB
-	UUID	Abschätzung als CLOB

SAP HANA	TINYINT	
-	SMALLINT	
-	INTEGER	
-	BIGINT	
-	SMALLDECIMAL	Abschätzung als FLOAT
-	DECIMAL	DECIMAL ohne Präzision und Skala wird als Float abgeschätzt
-	REAL	
-	DOUBLE	
-	DATE	
-	TIME	
-	SECONDDATE	Abschätzung als TIMESTAMP
-	TIMESTAMP	
-	CHAR	
-	VARCHAR	
-	NVARCHAR	
-	ALPHANUM	Abschätzung als NVARCHAR
-	SHORTTEXT	Abschätzung als NVARCHAR
-	CLOB	
-	NCLOB	
-	TEXT	Abschätzung als NCLOB
-	VARBINARY	
-	BLOB	
-	BOOLEAN	

4.4 Strukturen, Formate und Encodings

Der IM4-Archiver produziert bis zu drei verschiedene Dateiformate: Archiv-, Report- und Prüfsummen-Dateien. Alle Dateien sind UTF-8 codiert.

Die folgende Tabelle gibt eine Übersicht über das Encoding und die Formate der Dateien und Datentypen:

Typ	**Format oder Encoding**
Archiv	UTF-8 codierte JSON-Datei, evtl. GZIP-Kompression
Binär-Daten (BLOB, Binary, …)	Binär von der Db. empfangen und Base64 codiert
XML-Spalten/Dateien	Binär von der Db. empfangen und Base64 codiert
CLOB	UTF-8 Charakter, Base64 codiert
Time, Timestamps, Date, …	UTF-8 codiert, ISO 8601 Format
CHAR, VARCHAR, NCHAR, …	UTF-8 codiert
Numerische Datentypen	UTF-8 codiert

Eine Beschreibung der JSON Struktur der Archivdateien. Für unbekannte Datentypen ist der Wert der Records ein weiteres JSON Objekt.

```
Archive JSON:
{
        "DB_PROD": value,
        "DB_VERSION": value,
        "METADATA": [Schema-Metadata, ...],
        "CONTENT": [Schema-Content, ...]
}

Schema Metadata JSON:
{
        "Schema_NAME": value,
        "TABLES": [Table-Metadata, ...]
}

Table-Metadata JSON:
{
        "TABLE_INFO" : {
                "TABLE_CAT" : value,
                "TABLE_SCHEM" : value,
                "TABLE_NAME" : value,
                "TABLE_TYPE" : value
                        },
        "COLUMN_INFO": [Column, ...],
        "KEY_INFO": {
                "PRIM_KEY": [Primary Key, ...],
                "FOREIGN_KEY": [Foreign Key, ...]
                        },
        "INDEX_INFO" : [Index, ...]
}
```

```
Column JSON:
{
    "COLUMN_NAME" : value,
    "DATA_TYPE" : value,
    "TYPE_NAME" : value,
    "COLUMN_SIZE" : value,
    "DECIMAL_DIGITS" : value,
    "IS_NULLABLE" : value,
    "DATA_EST" : value
}


Schema-Content JSON:
{
        "SCHEMA_NAME" : value,
        "TABLES":       [Table-Content, ...]
}

Table-Content JSON:
{
        "TABLE_INFO" : {TABLE_INFO},
        "RECORDS" : [{"COLUMN_1":value, ...,
"COLUMN_N":value}, ...]
}
```