UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA DA COMPUTAÇÃO

Fabiano Niederauer Flôres

UM PROCESSO PARA A GERAÇÃO AUTOMATIZADA DE INSTÂNCIAS DE ESQUEMAS JSON A PARTIR DE CONSULTAS SQL

Santa Maria, RS
2019

Fabiano Niederauer Flôres

UM PROCESSO PARA A GERAÇÃO AUTOMATIZADA DE INSTÂNCIAS DE ESQUEMAS JSON A PARTIR DE CONSULTAS SQL

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC), da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**.

Orientador: Prof. Dr. Eduardo Kessler Piveta

Co-orientadora: Profª. Drª. Deise de Brum Saccol

Santa Maria, RS
2019

Flôres, Fabiano Niederauer
Um processo para a geração automatizada de instâncias de esquemas Json a partir de consultas SQL / Fabiano Niederauer Flôres.- 2019.
97 p.; 30 cm

Orientador: Eduardo Kessler Piveta
Coorientadora: Deise de Brum Saccol
Dissertação (mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Ciência da Computação , RS, 2019

1. Geração de Documentos Json 2. Esquemas Json 3. Modelo de dados I. Piveta, Eduardo Kessler II. Saccol, Deise de Brum III. Título.

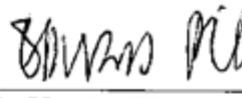
Sistema de geração automática de ficha catalográfica da UFSM. Dados fornecidos pelo autor(a). Sob supervisão da Direção da Divisão de Processos Técnicos da Biblioteca Central. Bibliotecária responsável Paula Schoenfeldt Patta CRB 10/1728.

Fabiano Niederauer Flôres

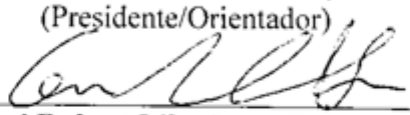
UM PROCESSO PARA A GERAÇÃO AUTOMATIZADA DE INSTÂNCIAS DE ESQUEMAS JSON A PARTIR DE CONSULTAS SQL

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC), da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**.

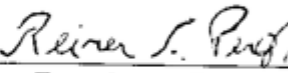
Aprovado em 28 de março de 2019:



Eduardo Kessler Piveta, Dr. (UFSM)
(Presidente/Orientador)



Giovani Rubert Librelotto, Dr. (UFSM)



Reiner Franthesco Perozzo, Dr. (UFN)

Santa Maria, RS
2019

DEDICATÓRIA

Dedico esta dissertação às minhas filhas, **Isabella Zanini Flôres** e **Beatriz Zanini Flôres**, que com amor e carinho me deram forças por todo o caminho percorrido, me motivando a procurar ser melhor a cada dia.

AGRADECIMENTOS

Agradeço ao apoio incondicional da minha família.

Às minhas filhas **Isabella Zanini Flôres** e **Beatriz Zanini Flôres**, por todo o amor e carinho, por seus lindos sorrisos e maravilhosos abraços que tanto me reconfortaram e trouxeram forças nos momentos difíceis.

À minha esposa **Alessandra da Silva Zanini**, por estar sempre ao meu lado, compartilhando sonhos e objetivos. O seu amor me fortalece a cada dia.

Aos meus pais **Luiz Antônio Feldman Flôres** (em memória) e **Zely Maria Niederauer Flôres**, por toda a dedicação, amor e bons exemplos.

Às minhas irmãs **Fabíola Niederauer Flôres** e **Fabrine Niederauer Flôres** por sempre me apoiarem.

Aos meus sogros **Eloi Pedro Zanini** e **Geísa da Silva Zanini** pelo carinho, atenção e apoio.

Aos meus orientadores **Prof. Dr. Kessler Piveta** e **Profª Drª Deise de Brum Saccol**, por toda a atenção e competência.

RESUMO

UM PROCESSO PARA A GERAÇÃO AUTOMATIZADA DE INSTÂNCIAS DE ESQUEMAS JSON A PARTIR DE CONSULTAS SQL

AUTOR: FABIANO NIEDERAUER FLÔRES
ORIENTADOR: EDUARDO KESSLER PIVETA
CO-ORIENTADORA: DEISE DE BRUM SACCOL

A crescente demanda por interoperabilidade entre sistemas modernos tem impulsionado o uso da notação JSON como um dos formatos preferidos para a troca de informações. Frequentemente, serviços *REST* são implementados para permitir a troca de informações entre esses sistemas. Vários bancos de dados relacionais se adaptaram a essa realidade fornecendo funções *SQL* estendidas que permitem o armazenamento e a manipulação de dados semiestruturados na notação *JSON*. No entanto, a falta de padronização entre os bancos de dados relacionais e a complexidade frequentemente observada nos esquemas exigidos dificulta a obtenção de documentos JSON originados de dados armazenados sob a ótica relacional. A correta desnormalização de dados, processo necessário para obter documentos semiestruturados originados de bancos de dados relacionais, requer conhecer a semântica das informações que serão extraídas. Através da análise dos esquemas JSON é possível mapear a estrutura semântica da informação que será representada. O objetivo principal deste trabalho é apresentar uma metodologia que permita a geração de documentos JSON, independentemente da complexidade estrutural dos esquemas, e do banco de dados relacional utilizado. Através do processo de análise automática de esquemas JSON e do uso de consultas *SQL* padrão, foi possível demonstrar, através de um estudo de caso, a eficácia da metodologia proposta.

Palavras-chave: Documentos *JSON*, Esquemas *JSON*, bancos de dados reacionais, bancos de dados orientados a documentos, consultas SQL.

ABSTRACT

A PROCESS FOR THE AUTOMATED GENERATION OF JSON SCHEMA INSTANCES FROM SQL QUERIES

AUTOR: FABIANO NIEDERAUER FLÔRES
ORIENTADOR: EDUARDO KESSLER PIVETA
CO-ORIENTADORA: DEISE DE BRUM SACCOL

The growing demand for interoperability among modern systems has driven the use of JSON notation as one of the preferred formats for information exchange. Frequently, *REST* services are implemented to allow the exchange of information between these systems. Several relational DBMSs have adapted to this reality by providing extended SQL functions that allow the storage and manipulation of semi-structured data in JSON notation. However, the lack of standardization between the relational DBMSs and the complexity frequently observed in the required schemas makes it difficult to obtain JSON documents originated from data stored from a relational perspective. The correct denormalization of data, a process necessary to obtain semi-structured documents originated from relational databases, requires knowing the semantics of the information that will be extracted. Through the analysis of the Esquemas JSON it is possible to map the semantic structure of the information that will be represented. The main objective of this article is to present a methodology that allows the generation of JSON documents, regardless of the structural complexity of the schemas, and the relational DBMS used. Through the process of automatic analysis of Esquemas JSON and the use of standard SQL queries, it was possible to demonstrate, through a case study, the effectiveness of the proposed methodology.

Keywords: JSON Documents, JSON schemas, reactional databases, document-oriented databases, SQL queries.

LISTA DE FIGURAS

LISTA DE TABELAS E QUADROS

LISTA DE ABREVIATURAS E SÍMBOLOS

NoSQL	*Not Only SQL*
SGBD	Sistema Gerenciador de Banco de Dados
REST	Representational State Transfer
SQL	*Structured Query Language*
JSON	*JavaScript Object Notation*
XML	*Extensible Markup Language*
YAML	*Ain't Markup Language*
W3C	*World Wide Web Consortium*
SGML	*Standard Generalized Markup Language*
HTML	*HyperText Markup Language*
DTD	*Document Type Definition*
XSD	*XML Schema Definition*
IETF	*Internet Engineering Task Force*
URI	*Uniform Resource Identifier*
ANSI	*American National Standards Institute*
BPMN	*Business Process Model and Notation*
IDE	*Integrated Development Environment*
LkJSON	JSON *Delphi library*
LAI	Lei de Acesso à Informação

SUMÁRIO

1 INTRODUÇÃO

Os avanços tecnológicos ocorridos nas últimas décadas aumentaram a demanda pela interoperabilidade entre sistemas, muitas vezes, desenvolvidos sob paradigmas distintos. Segundo Cafezeiro et al. (2008), interoperabilidade é a capacidade de sistemas computacionais operarem e cooperarem mesmo na presença de diferentes representações de dados e protocolos de comunicação. Em função das crescentes expectativas dos clientes em relação a novos produtos e novos avanços tecnológicos, o ciclo de vida dos produtos está ficando cada vez menor. Esse encurtamento do ciclo de vida exige que trocas de informações ocorram de forma mais veloz (DOS SANTOS, 2011). Face a este cenário, profissionais de tecnologia da informação vem sendo confrontados com novos desafios para o intercâmbio, tratamento, e armazenamento de dados.

Neste contexto, bancos de dados relacionais e não relacionais coexistem e necessitam trocar informações. Novos sistemas de bancos de dados vêm surgindo em resposta às novas demandas do mercado. Os sistemas de bancos de dados *NoSQL* (*Not Only SQL*) usam modelos de dados tais como: colunares, chave-valor, grafos e documentos (STRAUCH, 2011). Tais modelos possuem algumas vantagens em relação ao modelo relacional (JATANA et al. 2012). Porém, tradicionais bancos de dados relacionais gradativamente estão incorporando novas funções para o armazenamento e tratamento de dados semiestruturados (PETKOVIĆ, 2017). Possuir a capacidade de armazenar e tratar dados semiestruturados aproximam os bancos relacionais de seus concorrentes não relacionais. Bancos de dados *NoSQL* utilizam modelos de dados altamente versáteis e adaptativos, pois não necessitam de esquema preestabelecido. Na medida em que os bancos de dados relacionais disponibilizam novas formas de tratamento para dados semiestruturados, algumas vantagens antes existentes somente no modelo *NoSQL* deixam de ser destaque, tais como: a independência de esquema predefinido; e a capacidade de armazenar, de forma simplificada, dados de fontes heterogêneas.

O aumento exponencial do número de sistemas desenvolvidos para uso por meio da Internet, a geração de conteúdo por dispositivos móveis e o crescente número de pessoas e dispositivos conectados, potencializaram o surgimento de novas formas de intercâmbio de dados. Tais sistemas frequentemente necessitam transmitir e receber um grande volume de informações originadas de fontes distintas e que sofrem constantes mudanças. A demanda por interoperabilidade entre os sistemas modernos tem impulsionado o uso da notação JSON (*JavaScript Object Notation*) como um dos formatos preferidos para a troca de informações.

Diferente da *XML* (*Extensible Markup Language*), o JSON utiliza uma estrutura de arquivos composta de propriedade-valor e permite uma visível redução de bytes ao arquivo, possibilitando o intercâmbio de dados com uma menor exigência de banda (JÚNIOR et al., 2018). Frequentemente, serviços *REST* (*Representational State Transfer*) são implementados para permitir a troca de informações entre esses sistemas (ISSARNY, 2011). Em um estudo recente, realizado por PETKOVIĆ (2017), fora constatado que vários bancos de dados relacionais se adaptaram a essa realidade. Tais bancos de dados passaram a fornecer funções *SQL* (*Structured Query Language*) estendidas que permitem o armazenamento e a manipulação de dados semiestruturados na notação *JSON*. No entanto, a disparidade entre as novas funções incorporadas aos diferentes bancos de dados relacionais e a complexidade frequentemente observada nas consultas *SQL* exigidas, dificulta a obtenção de documentos JSON originados de estruturas relacionais.

Sistemas legados, muitos destes de grande porte, desenvolvidos e mantidos ao longo de vários anos sob a ótica relacional, atualmente necessitam transmitir dados semiestruturados para sistemas externos. Para estes sistemas legados, baseados em paradigmas distintos, a migração do banco de dados para uma versão recentes, muitas vezes é considerada inviável devido à alta complexidade de seus modelos. Para estes sistemas, a extração de informações semiestruturadas, originada de suas bases de dados, é ainda mais custosa. Tais sistemas demandam constantes implementações, específicas para cada novo esquema a ser instanciado. Alterações nas implementações existentes, sempre que surgem novas versões dos esquemas já implementados, normalmente envolvem alto custo de análise e desenvolvimento. Este trabalho propõem uma solução para a obtenção automatizada de dados semiestruturados na notação *JSON*, originados de bancos de dados relacionais. Através da análise automática dos esquemas, do processamento de algoritmos recursivos, e da utilização de consultas *SQL* padrão, o processo de obtenção dos documentos JSON se torna versátil. A solução proposta não necessita de novas implementações para se adaptar a novos esquemas JSON, independentemente da complexidade das estruturas que os compõem.

O objetivo deste trabalho é a definição de um método para a geração automatizada de documentos *JSON*. Por meio da vinculação de consultas *SQL* às propriedades estruturais dos esquemas JSON, e de campos da projeção destas consultas às propriedades simples destes esquemas, o processo resultará em instâncias exatas dos esquemas JSON. Os documentos JSON obtidos poderão ser utilizados para o armazenamento em coleções de bancos *NoSql* orientados a documentos, ou serem utilizados para a o intercâmbio de dados entre sistemas distintos. A solução proposta neste trabalho contribui para a obtenção de dados semiestruturados para

múltiplos propósitos, sendo a fonte dos dados um banco de dados Relacional moderno ou um banco de dados legado. Contribui ainda para a migração destes dados, para novas tecnologias de armazenamento. A tarefa de analisar os esquemas e mapeá-los, bem como a geração dos documentos *JSON*, ocorrerá de forma automatizada, desta forma cientistas de dados poderão concentrar seus esforços na confecção das consultas e na análise dos dados. A metodologia proposta necessitará de uma base de dados relacional "populada" (em uso). A validade da proposta desenvolvida será demonstrada por meio de um estudo de caso, no qual será utilizado um protótipo funcional desenvolvido para esta finalidade.

Este trabalho está organizado da seguinte forma: O capítulo 2 contextualiza temas importantes que servem de base para o entendimento do que foi proposto. Apresenta alguns trabalhos existentes que estão relacionados ou que possam contribuir, diretamente, à geração de dados estruturados e ou migração entre os modelos relacional e orientado a documentos; O capítulo 3 descreve em detalhes a proposta deste trabalho, definindo procedimentos para a obtenção dos resultados esperados; O capítulo 4 descreve os algoritmos desenvolvidos para o processamento das etapas da proposta deste trabalho; No capítulo 5 são descritos os experimentos realizados por meio do mapeamento automático de esquemas JSON, vinculação de consultas *SQL*, e geração de documentos JSON instanciados a partir dos esquemas mapeados. O capitulo 6 apresenta as conclusões sobre os resultados obtidos no processo de geração dos documentos JSON.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 MODELOS DE DADOS

Modelo de dados consiste na forma com que os dados são organizados em estruturas lógicas. A escolha de um modelo de dados depende do domínio das informações a serem dispostas, bem como dos objetivos que se deseja alcançar. Optar por um modelo implica em ponderar questões como: velocidade de consulta e escrita, versatilidade, redução de custos de desenvolvimento e manutenção, dentre outras. O modelo ideal para cada projeto é aquele que melhor se alinhar às prioridades do projeto.

Atualmente existem diversos modelos de dados, alguns dos mais conhecidos são: modelo relacional, modelo orientado a objetos, modelo semiestruturado e modelo de grafos; dois dentre estes, modelo relacional e semiestruturado, são de fundamental importância para a proposta desenvolvida neste trabalho, sendo estes descritos nos capítulos a seguir.

2.1.1 Modelo Relacional

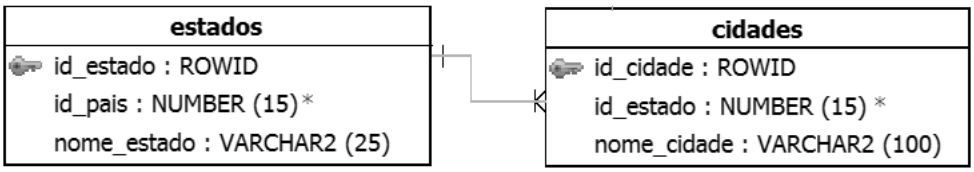
Fundamentado pela teoria dos conjuntos e na álgebra relacional, o modelo resultou de um estudo teórico apresentado por CODD (1970), e implementado na década de 80. No modelo relacional, os bancos de dados necessitam de um esquema predefinido para o armazenamento dos dados, indexação, processamento e otimização de consultas, e atualizações. Para garantir que todos os dados estejam em conformidade com o esquema de armazenamento definido, regras de integridade referencial são aplicadas.

O modelo relacional representa o banco de dados como uma coleção de relações (ELMASRI e NAVATHE, 2006). Tais relações são percebidas pelo usuário como tabelas bidimensionais, sendo o relacionamento entre estas tabelas definidos por meio de chaves estrangeiras. As tabelas do banco de dados consistem em colunas e linhas de informação, relações lógicas e restrições. A cada coluna é atribuído um tipo de dados, podendo ser numéricos, alfanumérico, binários, booleanos, entre outros. Regras de integridade são aplicadas para que cada coluna receba somente o tipo de dado definido.

2.1.1.1 Relacionamento entre Tabelas

As tabelas do modelo são vinculadas através de relacionamentos utilizando colunas chaves, tais chaves são únicas e identificam uma única linha da tabela relacionada. Dois tipos de chaves são definidos, chaves primárias e chaves estrangeiras. A figura 1 a seguir representa o relacionamento entre as tabelas estados e cidades.

Figura 1 – Relacionamento entre tabelas do modelo relacional



Fonte: Autor.

A coluna “id_estado” é chave primária da tabela estados, e chave estrangeira da tabela cidades. Desta forma um estado poderá se relacionar com várias cidades. Os relacionamentos entre tabelas possuem cardinalidade, esta poderá ser dos tipos: um para vários (1-n), um para um (1-1) ou vários para vários (n-n). Na ocorrência de cardinalidade vários para vários (n-n), deverá ser incluída nova tabela para resolver o relacionamento cruzado. Para garantir a consistência dos relacionamentos, regras de integridade referencial são aplicadas. Considerando o exemplo da figura 1, regras de integridade referencial impediriam que um estado fosse deletado caso este possua cidades referenciadas.

2.1.1.2 Normalização do Modelo

Normalização consiste em aplicar uma série de regras sobre as tabelas do modelo relacional, com o objetivo de verificar se o modelo de dados está corretamente concebido. A normalização frequentemente é realizada até a 3ª forma normal, esse processo é realizado para tornar o armazenamento dos dados mais eficiente, eliminando redundância e inconsistência (SETZE; SILVA, 2005).

Existem cinco regras de normalização, conhecidas como formas normais, são elas:

- 1º Forma Normal (1FN) – Cada linha da tabela deverá corresponder a apenas um registro, e cada coluna deverá corresponder a somente um campo. A figura a seguir ilustra a transformação da tabela após a aplicação da 1º forma normal:

Quadro 1 - Aplicação da 1º forma normal

Antes da aplicação da 1FN

enderecos					
pais	**estado**	**cidade**	**rua**	**numero**	**complemento**
Brasil	RS	Santa Maria	R. Floriano Peixoto	112	ap 120
			R. Benjamin Constant	1000	
			Av. Rio Nranco	325	
		Canoas	R. Brasil	2587	ap 305
		Porto Alegre	Av. Voluntários da Pátria	1152	
	SC	Florianópolis	Av. Madre Benvenuta	45	casa 2
	SP	São Paulo	Av Paulista	485	

Resultado após a regra 1FN

enderecos					
pais	**estado**	**cidade**	**rua**	**numero**	**complemento**
Brasil	RS	Santa Maria	R. Floriano Peixoto	112	ap 120
Brasil	RS	Santa Maria	R. Benjamin Constant	1000	
Brasil	RS	Santa Maria	Av. Rio Nranco	325	
Brasil	RS	Canoas	R. Brasil	2587	ap 305
Brasil	RS	Porto Alegre	Av. Voluntários da Pátria	1152	
Brasil	SC	Florianópolis	Av. Madre Benvenuta	45	casa 2
Brasil	SP	São Paulo	Av Paulista	485	

Fonte: Autor

No exemplo a cima, a aplicação da 1º forma normal resultou em redundância de informação. Devido a este fato algumas anomalias poderão ocorrer: Ao inserir um registro poderia ocorrer vários campos com valor nulo; ao editar um campo, poderia ser necessário editar também o mesmo em outros registros; e ao deletar um campo, poderia ser necessário deletá-lo de outros registros.

- 2º Forma Normal (2FN) – Antes de aplicar a 2FN, a tabela tem que estar na 1FN e cada atributo não chave tem que ser funcionalmente dependente da chave primária. O exemplo a seguir demonstra a tabela originalmente denominada “endereços” sendo desmembrada em 4 tabelas para atingir a 2FN.

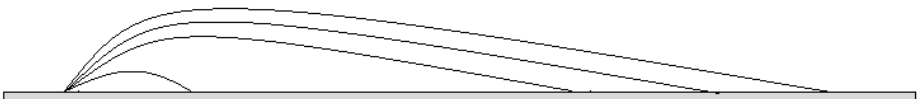
Quadro 2 – Aplicação da 2º forma normal

Fonte: Autor.

O exemplo demonstra que os problemas potenciais que existiam na 1FN, deixaram de existir na 2FN. Qualquer operação de inserção, edição e exclusão de um determinado campo não chave seria possível alterando um único registro.

- 3º Forma Normal (3FN) - A tabela deve estar na 2FN e nenhum atributo não chave pode depender funcionalmente de outro atributo que não seja a chave primária. Caso existir alguma dependência funcional entre atributos não chave, então, é necessário retirar esse conjunto de atributos da tabela e construir com eles uma tabela à parte.

Quadro 3 – Aplicação da 3º forma normal



Endereço						
id_endereco	id_cidade	cod_lograd	decr_lograd	rua	numero	complemento
1	1	R	Rua	R. Floriano Peixo	112	ap 120
2	1	R	Rua	R. Benjamin Cor	1000	
3	1	Av	Avenida	Av. Rio Branco	325	
4	2	R	Rua	R. Brasil	2587	ap 305
5	3	Av	Avenida	Av. Voluntários	1152	
6	4	Av	Avenida	Av. Madre Benv	45	casa 2
7	5	Av	Avenida	Av Paulista	485	

Após aplicar a 3FN

Endereço					
id_endereco	id_cidade	id_tipo_lograd	rua	numero	complemento
1	1	1	R. Floriano Peixo	112	ap 120
2	1	1	R. Benjamin Cor	1000	
3	1	2	Av. Rio Branco	325	
4	2	1	R. Brasil	2587	ap 305
5	3	2	Av. Voluntários	1152	
6	4	2	Av. Madre Benv	45	casa 2
7	5	2	Av Paulista	485	

id_tipo_lograd

tipos_lograd		
id_tipo_lograd	cod_lograd	decr_lograd
1	R	Rua
2	Av	Avenida

Fonte: Autor.

A tabela endereços representada no exemplo a cima possui as colunas “cod_lograd” e “descr_lograd” dependentes entre si. Para aplicar a 3FN foram retirados para uma tabela própria denominada “tipos_lograd”. O relacionamento entre as tabelas endereços e “tipos_lograd” fora preservada utilizando a chave “id_tipo_lograd”.

- 4º Forma Normal (4FN) – A tabela deverá estar na 3FN e não possuir dependências multivaloradas. No exemplo a seguir as colunas dependem umas das outras.

Quadro 4 – Aplicação da 4º forma normal

Antes da 4FN

paciente_exames		
id_paciente	id_plano	id_exame
1	1	1
2	1	1
1	1	2
2	2	1
2	3	1
2	2	2
3	5	2

Resultado após 4FN

paciente_planos	
id_paciente	id_plano
1	1
2	1
2	2
2	3
3	5

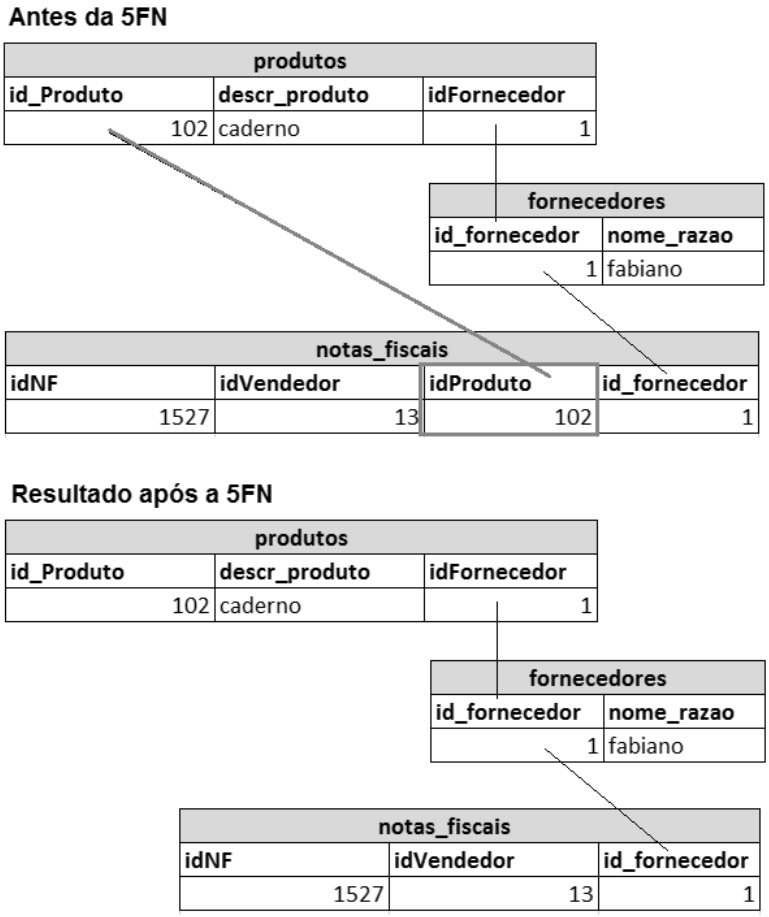
paciente_exames	
id_paciente	id_exame
1	1
2	1
1	2
2	2
3	2

Fonte: Autor.

Para a 4FN as colunas foram separadas em duas tabelas distintas, resolvendo assim a dependência múltipla.

- 5º Forma Normal (5FN) – A tabela deverá estar na 4FN e não possuir relacionamentos múltiplos (ternários, quaternários e n-ários).

Quadro 5 – Aplicação da 5º forma normal



Fonte: Autor.

Neste exemplo, a tabela “notas_fiscais” originalmente possuía a coluna “idProduto” como chave estrangeira. Porém observa-se a existência da mesma chave na tabela fornecedores, também referenciada por “notas_fiscais”. Para obter a 5FN fora retirado o relacionamento direto que havia entre “notas_fiscais” e “produtos”.

2.1.2 Modelo Semiestruturado

Atualmente novos serviços, para atender às mais variadas demandas, vem sendo disponibilizados para acesso via internet, principalmente envolvendo tecnologias para dispositivos móveis. Este movimento aumentou a necessidade de geração e transmissão de informações entre sistemas distintos, muitos destes concebidos no modelo de dados relacional, para novas tecnologias de armazenamento.

O Modelo Semiestruturado consiste numa coleção de dados heterogêneos, que não possuem uma estrutura rígida (ASAI, et al., 2004). Entre os padrões comumente utilizados para a troca de informações entre tais sistemas, os documentos na notação XML (eXtensible Markup Language), YAML (acrónimo para Ain't Markup Language) e JSON (JavaScript Object Notation) são os que mais se sobressaem, principalmente quando a transmissão se dá por meio de Web Services. Tanto o padrão XML como JSON são notações que permitem grande flexibilidade estrutural, sendo facilmente adaptáveis para a utilização em praticamente todos os escopos. Outra vantagem que colabora para a maior utilização destes padrões é o amplo suporte provido pelas principais plataformas de desenvolvimento atuais.

Embora o formato XML, tradicionalmente consolidado como um padrão na transmissão de dados, demonstre grande eficiência; a notação JSON nos últimos anos vem se destacando pela simplicidade e menor volume de dados necessários à transmissão. Frente a este senário, a conversão e geração de dados semiestruturados no formato JSON cada vez mais tem sido exigida. Para sistemas legado, muitos destes utilizando tecnologias que não possuem suporte a estes formatos, a geração destes artefatos consiste em um desafio a ser superado.

2.1.2.1 XML (Extensible Markup Language)

A *XML* foi criada e padronizada pela *W3C* (*World Wide Web Consortium*) como um subtipo da *SGML (Standard Generalized Markup Language)*. O propósito inicial foi simplificar o compartilhamento de dados através da internet, no entanto, a *XML*, por combinar a flexibilidade da *SGML* com a simplicidade da *HTML (HyperText Markup Language)*, logo passou a ser considerada a linguagem padrão para o intercâmbio de informações entre aplicações distintas, ou seja, um arquivo que contenha dados no padrão *XML*, gerado por uma dada aplicação, pode ser lido e compreendido por outra aplicação de banco de dados distinta.

A *XML* tornou-se uma forma muito utilizada para armazenar dados de sistemas e trocar informações entre aplicações, principalmente entre as aplicações que têm interfaces disponíveis na Internet. Devido à maturidade, ampla usabilidade, e alta capacidade de serialização, tem sido considerada formato padrão para o transporte de informações (GOLDBERG, 2009).

Tendo a portabilidade como uma das principais características, a *XML* é um formato que independe de plataforma de hardware e software. A *XML* possibilita a criação de estruturas de dados hierarquicamente organizadas. A estrutura de um documento *XML* depende do tipo de informação que será manipulada e da forma com que tais informações foram modeladas. A modelagem de um documento no padrão *XML* depende do domínio da aplicação, sendo assim,

os elementos que compõem as estruturas variam de acordo com a necessidade de cada aplicação.

De acordo com AZEVEDO et. al (2005), *XML* é uma sintaxe para transmissão de informações que permite a especificação dos dados de forma a facilitar seu intercâmbio e reutilização por múltiplas aplicações.

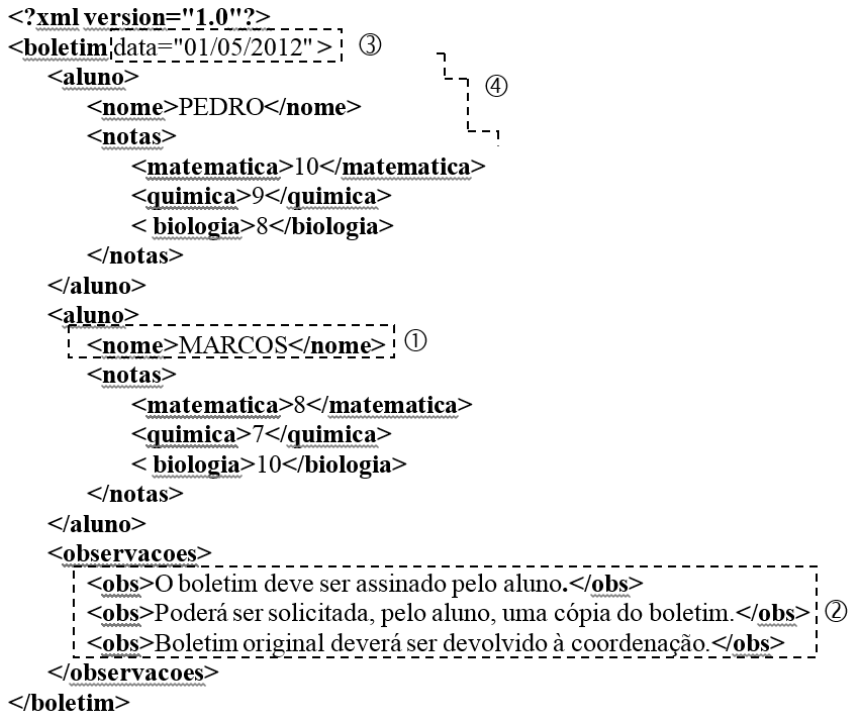
Os principais conceitos inerentes ao padrão *XML* foram delineados por AZEVEDO et. al (2005). São estes:

- **Elemento -** Estruturas em *XML* são representadas em forma textual, tendo como componente básico o elemento. Elementos são segmentos de texto delimitado por marcadores, no interior de um elemento pode haver texto simples, outros elementos ou ambos. Como exemplo, na figura 1, item ①, observa-se o marcador de início <nome> e o marcador de final </nome>, tudo que estiver entre os marcadores é considerado conteúdo. Um conjunto de marcadores idênticos forma uma coleção, conforme se observa na figura 1, item ②.

- **Atributo -** Os atributos são usados para descrever os elementos, indicando suas propriedades. Os valores dos atributos são inseridos como sequências de caracteres conforme é possível observar na figura 1, item ③.

- **Contexto -** O contexto de um conteúdo é definido como sendo sua posição na árvore que representa o documento. É representado pelo caminho ou sequência de marcadores que se inicia no elemento raiz e termina no elemento que o contém, observa-se um exemplo na figura 1, item ④.

Figura 2 - Estrutura básica de um documento *XML*



Fonte: Autor.

Documento *XML* ao serem confeccionados devem respeitar estritamente algumas regras de sintaxe, a observação a essas regras o qualificam como documento válido segundo os padrões desta notação. Dois validadores do padrão *XML* amplamente utilizados são o *DTD (Document Type Definition)* e *XSD (XML-Schema Definition)*. O *DTD*, surgiu com o primeiro validador para documentos *XML*. Devido a algumas carências de tipos de dados incompatíveis e o fato de que a própria sintaxe do validador é distinta do padrão *XML*, o *DTD* fora gradativamente substituído pelo *XSD*, as principais vantagens trazidas pelo *XSD* contribuíram para a consolidação do *XML* como solução de referência para o intercâmbio de dados, as principais vantagens em relação a seu antecessor *DTD* são: *XSD* são esquemas escritos em *XML*; *XSD* é extensível, permitindo assim edições futuras; e suporte a uma maior variedade de tipos de dados.

Os esquemas *XML* podem conter as definições de tipo, tamanho, ocorrência, regras de preenchimento, e demais informações dos elementos que compõem os documento *XML*. Esquemas *XML* são responsáveis por impor restrições nos formatos e valores que os documentos *XML* podem expressar. Documento *XML* possui referências que identificam os esquemas utilizados, porém os esquemas *XML* não "conhecem" quantos documentos *XML* o referenciam. Ao conjunto formado por um esquema *XML* e um conjunto de documentos com

referência a este esquema, dá-se o nome de base de dados semiestruturados (COHEN et al., 1998).

Documentos *XML* podem referenciar esquemas disponibilizados em uma rede de dados, o conjunto formado por um esquema e um documento *XML* possui aspecto distribuído (SILVEIRA, 2007).

Ainda segundo SILVEIRA (2007), os dois principais formatos para representação de esquemas *XML* são:

- *DTD* (*Data Type Definition*) é um formato de representação de esquemas que restringe-se a possibilitar uma estruturação básica dos documentos *XML*, definindo a estrutura hierárquica da informação, porém não permite definir os valores contidos nos elementos que compõem tais documentos. Esquemas no formato *DTD* não são representados no padrão da *XML*;

- *XML Schema* além de permitir a estruturação hierárquica dos documentos *XML*, também permite, por meio do uso de expressões regulares, expressar as regras de validação dos valores dos elementos e atributos que compõem os documentos. *XML Schema*, ao contrário do *DTD*, é representado respeitando o padrão *XML*.

Pela natureza dinâmica dos sistemas de informação, documentos representados no padrão *XML* e esquemas nos formatos *DTD* e *XML Schema*, podem sofrer alterações, estando sujeitos as necessidades de evolução. Embora amplamente adotados, documentos *XML* apresentam sintaxe redundante e demandam maior espaço de armazenamento em relação aos padrões YAML (BEN-KIKI et al., 2009) e o JSON (CROCKFORD, 2006), que em vários casos têm sido utilizados como um substituto ou como um complemento do *XML*.

2.1.2.2 YAML (Ain't Markup Language)

Proposto em 2001, é um formato de serialização concebido para a representação de dados por combinações de listas, *hashes* e valores simples e escaláveis. Inspirado nos conceitos de linguagens como *C*, *Perl*, *Python*, e *XML*, além de possuir sintaxe legível a humanos e de fácil mapeamento, possuir estruturação simples, e de não necessita de frequentes conversões entre tipos de dados, já que suporta a maioria dos tipos mais utilizados nas plataformas e linguagens de programação.

A notação baseia-se em endentação formada exclusivamente por espaços em branco, não admitindo tabulações. A estrutura de um documento na notação *YAML* é formada basicamente por listas e dicionários compostos por uma coleção de elementos chave e valor. A figura a seguir exemplifica um documento na notação *YAML* e destaca os artefatos estruturais de lista e dicionário.

Figura 3 - Estrutura básica de um documento *YAML*

```
version: "1.0"
boletim:
  data: "01/05/2012"
  nome: [pedro, marcos]
  pedro:
    notas: {matematica: 10, quimica: 9, biologia: 8}
  marcos:
    notas: {matematica: 8, quimica: 7, biologia: 10}
  observacoes: ["O boletim deve ser assinado pelo aluno", "Poderá ser solicitada, pelo
aluno, uma cópia do boletim", "Boletim original deverá ser devolvido à coordenação"]
```

Fonte: Autor.

No exemplo acima é possível observar as listas boletim, nome, observações; e as coleções notas.

2.1.2.3 JSON (JavaScript Object Notation)

A notação JSON possibilita a composição de artefatos com estruturação leve e simplificada. Ao compará-lo com XML, e notória a vantagem de que em JSON não ocorrem redundâncias de tags, tornando menor o volume de dados e, consequentemente, reduzindo a quantidade de memória necessária para o seu armazenamento, manipulação e transmissão. Por possibilitar instanciar documentos menores, com estruturas simples e de fácil leitura, a notação JSON fora amplamente adotada, principalmente como formato escolhido para a transmissão de dados entre aplicações via Internet.

O JSON é uma notação para a composição de documentos textuais contendo dados estruturados. A notação provém dos *object literals* do *JavaScript*, sua sintaxe é formada basicamente por uma estrutura hierárquica composta pelos artefatos estruturais de vetores, matrizes e objetos, e pares de elementos nome e valor, estes dos tipos primários de cadeias de caracteres, números, booleanos e nulos. Assim como em *JavaScript*, JSON representa os valores de forma simples, atribuindo a eles nomes que os identificam. Embora seja se assemelhe

com a sintaxe de *object literals* do *JavaScript*, JSON é independente de linguagem de programação.

A estruturação de um documento na notação JSON é composta de forma hierárquica por uma sequência de vetores, matrizes e objetos. As matrizes são estruturas multidimensionais delimitadas pela abertura e fechamento de colchetes, sendo seus elementos separados por vírgulas. Tal como ocorre com as matrizes, os vetores são delimitados por colchetes e seus elementos também separados por vírgulas, porém os vetores são estruturas unidimensionais. Já os objetos são delimitados pela abertura e fechamento de chaves, admitem múltiplos pares de elementos nome e valor, assim como matrizes e até mesmo outros objetos. Objetos permitem estruturas complexas podendo representar informações das mais variadas formas e contextos.

Exemplo de Vetor

```
["RJ", "SP", "MG", "ES"]
```

Exemplo de Matriz Bidimensional

```
[
  [1,5],
  [-1,9],
  [1000,0]
]
```

Exemplo de Objeto

```
{
  "titulo": "Objeto na Notacao JSON",
  "resumo": "Objetos sao representados em JSON desta forma",
  "ano": 2019,
  "palavras_chave": ["JSON", "semiestruturado", "dados"]
 }
```

Informações são representadas por um conjunto de pares nome e valor, sendo que o nome é delimitado pela abertura e fechamento de aspas duplas, seguido de dois pontos e do respectivo valor. Os valores poderão ser de tipos primários, são eles:

- Cadeia de Caracteres (*string*)

Valores do tipo *string* são representados entre aspas duplas.

```
"nome" : "Fabiano N Flores"
```

- Numéricos (*Number*)

Valores numéricos poderão se inteiros ou conterem casas decimais. Números decimais utilizam o caractere ponto(.) antes da parte fracionada. Números negativos são definidos pela precedência do caractere menos (-).

```
“idade“ : 40
“altura“ : 1.85
“código” : -2
```

- Booleanos (*Boolean*)

Valores booleanos são definidos por *true* (verdadeiro) ou *false* (falso)

```
“casado” : true
```

- Valores Nulos (*Null*).

A palavra reservada *null* deverá sempre ser utilizada para a representação da não atribuição de valor para determinado elemento.

```
“peso”: null
```

O exemplo a seguir ilustra um documento JSON formado pelos tipos estruturais e primários descritos acima:

```
{
   "version":"1.0",
   "boletim":{
      "data":"01/05/2012",
      "alunos":[
         {
            "nome":"pedro",
            "notas":{
               "matematica":10,
               "quimica":9,
               "biologia":8
            }
         },
         {
            "nome":"marcos",
            "notas":{
               "matematica":8,
               "quimica":7,
               "biologia":10
            }
         }
      ],
      "observacoes":[
         "O boletim deve ser assinado pelo aluno",
         "Poderá ser solicitada, pelo aluno, uma cópia do boletim",
         "Boletim original deverá ser devolvido à coordenação"
      ]
   }
}
```


O JSON é uma escolha típica em ambientes onde o fluxo de dados entre o cliente e o servidor é de grande importância, onde a fonte dos dados tem de ser confiável e não pode existir perda dos recursos de processamento do lado do cliente (para manipulação de dados ou geração de interface). Este é o tipo de ambientes onde os grandes players tecnológicos (*Google*, *Yahoo*, entre outros) estão inseridos sendo esta a razão pela qual utilizam este formato (SANTOS, 2016).

2.2 ESQUEMA JSON

O esquema JSON é um vocabulário que define a forma de notação e validação de documentos JSON. Este capítulo foi escrito em conformidade com as especificações descritas em recentes documentos de trabalho do *IETF (Internet Engineering Task Force)*, disponíveis para acesso em *JSON-schema.org*.

O esquema JSON destina-se a definir validação, documentação, navegação de *hiperlink* e controle de interação de dados *JSON*. A sintaxe do esquema consiste na definição das palavras reservadas (vocabulário) que definem validação, vinculação, anotação, navegação, interação e referências entre outros esquemas (WRIGHT et al., 2018). O esquema JSON é um documento textual concebido na notação JSON. Documentos JSON validados contra um esquema JSON são denominados de instâncias, sendo que uma instância é um conjunto de tipos estruturais e tipos primitivos como matrizes, objetos, cadeias de caracteres, numéricos, booleanos e campos nulos. A seguir estão descritas algumas regras básicas previstas pelo esquema JSON:

- Espaços em branco não são considerados: Embora possam haver endentação para melhor visualização por humanos, o esquema ignora quaisquer ocorrências destes espaços;

- Zeros à direita são ignorados: A ocorrência de zeros a direita em valores numéricos não é considerada pelo esquema; e

- Nomes (rótulos) não se repetem no escopo de um mesmo objeto: Em um mesmo objeto não é permitido que haja dois nomes idênticos.

As propriedades que são usadas para descrever o esquema são denominadas palavras-chave do esquema, tais palavras-chave são definidas pelo vocabulário definido para o esquema.

Para atender requisitos específicos de cada implementação, os esquemas JSON poderão ser estendidos adicionando novas palavras-chave. Estas implementações devem referenciar o esquema original utilizando a palavra-chave *"$schema"*. O valor desta palavra-chave deve ser um *URI (Uniform Resource Identifier)*, que consiste em um localizador normalizado, utilizado para a identificação e localização do esquema original contra o qual o esquema estendido poderá ser validado.

2.2.1 Esquema raiz e subesquemas

Esquema raiz (*root*) é o esquema que contém todos os subesquemas que compõem o documento. Subesquemas possuem palavras-chave específicas que os identificam, assim é

possível que diversos esquemas sejam dispostos de forma hierárquica. O exemplo a seguir ilustra o esquema raiz e o subesquema conteúdo.

```
{
   "titulo":"raiz",
   "conteudo":{
      "num_paginas":120,
      "resumo":"resumo "
   }
}
```

2.2.2 Referências entre Esquemas

As referências entre esquemas evitam a redundância de definições tornando o documento mais compacto. Porém o custo de processamento para a análise do esquema é aumentado devido à necessidade de resolver as referências previamente. Para a inserção de referências em esquemas *JSON* é utilizada a palavra-chave *"$ref"*, sendo o valor um *URI* que ao ser resolvido contra a *URI* do objeto raiz definirá o esquema a ser utilizado. Ao definir as referências do esquema, deve-se garantir que referências recursivas não resultem em laços infinitos. Referências recursivas são possíveis através de auto-referenciamento, quando um esquema referencia a si próprio; ou quando dois esquemas referenciam um ao outro. O exemplo a seguir ilustra a ocorrência de referência em um esquema JSON.

```
{
   "$id":"http://example.net/root.json",
   "pessoa":{
      "type":"matriz",
      "itens":{
         "$ref":"#est_civil"
      }
   },
   "definitions":{
      "est_civil":{
         "$id":"#est_civil ",
         "type":"object",
         "additionalProperties":{
            "$ref":"other.json"
         }
      }
   }
}
```

Quando uma implementação encontra o esquema <#/definitions/est_civil>, ela resolve a referência *URI* "$id" em relação ao *URI* base atual para formar *<http://example.net/root.JSON# est_civil>*.

Quando uma implementação procura dentro do esquema *<#/est_civil>*, ela encontra a referência *<#est_civil>*, e resolve isso para *<http://example.net/root.JSON#est_civil>*, que ela viu definida neste mesmo documento e, portanto, pode usar automaticamente.

Quando uma implementação encontra a referência "*other.JSON*", ela resolve isso para *<http://example.net/other.JSON>*, que não está definido neste documento. Se um esquema com esse identificador tiver sido fornecido para a implementação, ele também poderá ser usado automaticamente.

Segundo WRIGHT et al. (2018) o *URI* não é um localizador de rede, e sim apenas um identificador. Um esquema não pode ser carregado por meio do *URI*, implementações não deve executar uma operação de rede quando encontram um *URI*, mesmo que endereçável pela rede, as implementações que utilizam esquemas JSON deverão carregar antecipadamente os esquemas que serão utilizados.

2.2.2.1 Reutilização de definições de esquemas

As palavras-chave "*definitions*" define um local padronizado para que as definições dos objetos que compõem o esquema sejam inseridas. O valor de "*definitions*" deve ser um objeto composto por subesquemas válidos. Desta forma os subesquemas definidos poderão ser reutilizados dentro do escopo do objeto principal ou raiz. O exemplo descrito a seguir representa a reutilização de um subesquema.

```
{
   "type":"array",
   "items":{
      "$ref":"#/definitions/inteiro_positivo"
   },
   "definitions":{
      "inteiro_positivo":{
         "type":"integer",
         "exclusiveMinimum":0
      }
   }
}
```

No exemplo, um esquema descreve uma matriz de inteiros, onde a restrição de inteiro positivo é um subesquema encontrado em *"definitions"*.

2.2.3 Validação de Instância JSON

O vocabulário define as palavras-chave usadas para garantir que um determinado documento *JSON* satisfaça determinado número de critérios. Palavras-chave específicas são definidas para possibilitar a geração de instâncias da interface com o usuário. Desta forma a validação de documentos contra o esquema JSON impõem restrições na estrutura dos dados. As restrições são definidas por quaisquer palavras-chave que contenham informações de asserção. Uma determinada instância é considerada válida se todos os seus elementos satisfazem todas as restrições declaradas.

A validação é iniciada confrontando o objeto raiz do documento JSON contra ao esquema instanciado. Subsequentemente, palavras-chave, específicas para esta finalidade, determinam os subesquemas que serão validados. Essas palavras-chave também definem se e como os resultados de asserção de subesquema são modificados ou combinado. Segundo WRIGHT et al. (2018) essas palavras-chave não impõem condições restritiva, sendo utilizadas exclusivamente para o controle das asserções aplicadas e avaliadas.

2.2.3.1 Vocabulário da Notação JSON

Os elementos que compõem documentos JSON são validados de acordo com critérios definidos por palavras-chave que integram o vocabulário utilizado. A seguir estão descritas as palavras-chave do esquema JSON proposto nos mais recentes estudos da *IETF*.

Palavras-chave de uso geral:

- *title* - O valor desta palavra-chave deverá ser uma cadeia de caracteres que poderá definir o título de um esquema raiz ou de um subesquema.
- *description* - O valor deverá ser do tipo cadeia de caracteres e poderá ser utilizado para informar o propósito da instância descrita pelo esquema.
- *type* – Esta palavra-chave identifica o tipo de dado ou o tipo do elemento estrutural o qual o elemento que a contenha se refere. Os valores válidos segundo o esquema são: *array*, *object*, *number*, *string*, *boolean ou null*;

- *enum* – O valor deverá ser um vetor contendo elementos de quaisquer tipos, admitindo inclusive elemento nulo. Uma instância desta definição é considerada válida se seu valor estiver previsto dentre os elementos deste vetor;
- *const* – Assim como na palavra-chave "*Enum*", os valores poderão ser de quaisquer tipos. Porém para ser considerada válida, uma instância deverá ter o valor idêntico quando comparado ao valor constante no esquema.
- *format* – É uma palavra-chave que poderá ser utilizada como uma anotação para transmitir valor semântico a uma instância, ou para definir um formato especifico para o valor de determinada propriedade. Ao definir um formato para os valores de determinadas propriedades, implementações poderão validar o referido formato utilizando o valor de "*format*" definido no esquema. Os principais formatos são: *date, time, date-time, email, hostname,* entre outros. Também é possível definir novos formatos para atender as necessidades específicas de cada implementação, desta forma estes novos formatos serão validados segundo as regras e restrições definidas nestes esquemas personalizados.

Aplicáveis a instâncias numéricas:

- *multipleOf* - Defini que valor numérico da instância seja múltiplo do valor informado para esta palavra-chave. A validade de uma instância é confirmada se a divisão de seu valor pelo valor determinado por *multipleOf* resulte em um número inteiro;
- *maximum* e *exclusiveMaximum* – Ambos determinam o limite superior que uma instância numérica poderá possuir. Porém *exclusiveMaximum* requer que o valor da instância seja inferior ao valor determinado;
- *minimum* e *exclusiveMinimum* – Determinam o limite inferior que poderá ser assumido por uma instância. A palavra-chave *exclusiveMinimum* requer que o valor da instância seja superior ao valor definido no esquema.

Aplicáveis a sequência de caracteres (Strings):

- *maxLength* – Define o comprimento máximo que um valor de instância do tipo cadeia de caracteres poderá possuir;

- *minLength* – Determina o comprimento mínimo que o valor da instância deverá possuir. Caso esta palavra-chave seja omitida, o comportamento esperado é que seja considerado como de valor zero;
- *pattern* – Esta palavra-chave deverá conter como valor uma expressão regular que restringe a cadeia de caracteres a determinada formatação (máscara). A sintaxe para este valor é um subconjunto das expressões regulares do *JavaScript* (TERLSON et al., 2018).

Aplicáveis a matrizes (Arrays):

- *items* e *additionalItems* - O valor desta palavra-chave deve ser um subesquema ou outra matriz de subesquemas válidos. Caso "*items*" seja um subesquema, instâncias serão consideradas válidas se os elementos da matriz forem validos contra esse subesquema. Caso "*items*" seja uma matriz de subesquemas, estes são considerados válidos se cada elemento destes atenderem as restrições definidas em "*additionalItems*";
- *maxItems* – O valor desta palavra-chave delimita a quantidade máxima de elementos de uma matriz;
- *minItems* - Delimita a quantidade mínima de elementos de uma matriz. Quando esta palavra-chave é omitida terá o mesmo efeito que um valor igual a 0;
- *uniqueItems* - O valor correspondente a esta chave deverá ser um booleano. Caso o valor seja "*true*" (verdadeiro), uma instância deste esquema será considerada válida se todos os elementos forem únicos dentre todos os elementos da matriz;
- *contains* – Esta palavra-chave deverá definir um subesquema *JSON* válido. Uma instância de matriz será considerada válida se ao menos um de seus elementos for válido em comparação com o subsquema definido em "*contains*".

Aplicáveis a Objetos (objects):

- *maxProperties* – Determina a quantidade máxima de propriedades que determinado objeto deverá conter;
- *minProperties* – Define a quantidade mínima de propriedades esperadas para determinado objeto. Se esta palavra-chave for omitida, o valor considerado será zero;

- *required* - O valor desta palavra-chave deve ser uma matriz contendo elementos do tipo cadeia de caracteres (*string*) exclusivos. Para considerar o esquema válido, uma instância deste objeto deverá possuir todas as propriedades cujo nome esteja contido na matriz definida por "*required*";
- *properties* - O valor para esta palavra-chave deve ser do tipo objeto (*object*). Todos os valores constantes neste objeto deverão ser esquemas válido, determinando como as instâncias secundárias serão validadas. A omissão de "*properties*" equivale a um objeto vazio;
- *patternProperties* - O valor de desta palavra-chave deve ser um objeto, sendo que cada nome das propriedades deste objeto deve estar escrita como uma expressão regular (TERLSON et al.; 2018). Os valores de todas as propriedades deste objeto devem ser esquemas válidos.

  A validação é confirmada se, para cada instância cujo nome corresponde a uma das expressões regulares em "*patternProperties*", o seu valor é confrontado com o esquema valor atribuído à expressão regular;
- *additionalProperties* - O valor deve ser um esquema válido que determina como as instâncias secundárias, "filhas" do objeto, serão validadas. A validação com "*additionalProperties*" aplica-se apenas aos valores de nomes de instâncias "filhas" que não correspondem a nenhum nome definido em "*properties*", e não corresponde a nenhuma expressão regular definida em "*patternProperties*";
- *propertyNames* - O valor desta palavra-chave deve ser um esquema válido. O esquema definido em "*propertyNames*" determina os nomes que as propriedades de determinada instância de objeto deverão possuir.

As palavras-chave relacionadas neste capítulo forma descritas por serem as mais relevantes. O vocabulário completo proposto pela *IETF* pode ser acessado por meio do site *JSON-schema.org.*

2.2.4 Condições de Interoperabilidade

Uma propriedade do tipo cadeia de caracteres (*string*) poderá possuir o valor "*null*" e mesmo assim ser considerada pelo esquema como uma instância válida. As implementações que farão uso do esquema JSON deverão estar aptas a trabalhar com valores nulos.

Instâncias numéricas possuem precisão arbitrária. O esquema não impõe restrições quanto ao limite de casas decimais. Portanto propriedades numéricas de determinada instância poderão apresentar valores longos. Implementações deverão ser capazes de processar tais valores.

2.3 ESTADO DA ARTE

A obtenção de documentos no formato JSON a partir de dados originados do modelo relacional, tornou-se de grande relevância na medida em que JSON passou a assumir um lugar de destaque como um dos principais formatos de intercâmbio de informações. Reconhecido como um formato leve, e de fácil compreensão por humanos, JSON vem ganhando espaço até então dominado pelo formato *XML*. Recentemente, uma nova era de desenvolvimento de aplicativos está surgindo, baseada na facilidade de acesso a recursos de computação modernos, como dispositivos móveis. Esse acesso pode ser suportado usando JSON. Portanto, o suporte de armazenamento e acesso de consulta para documentos JSON, no contexto dos bancos de dados relacionais, faz-se necessário. Por esse motivo, no ano de 2016 o comitê de padronização do *SQL* publicou uma proposta chamada *SQL* / *JSON*.

Atualmente muitos dos mais consagrados bancos de dados Relacionais integraram funções nativas para o armazenamento e a manipulação de dados no formato JSON. Em um estudo recente, PETKOVIĆ (2017) comparou alguns bancos de dados quanto às implementações padronizadas na *ANSI SQL/JSON*, o estudo destacou o banco de dados *Oracle* como sendo um dos que mais evoluiu em relação ao padrão *SQL/JSON*. A documentação das funções para a manipulação de JSON *Documents* poderá ser acessada em *Oracle Database online Documentation*.

A tarefa de geração de documentos JSON é constantemente atribuída a funções implementadas diretamente no código fonte dos sistemas. Funções fixadas em código e específicas para cada Esquema JSON a ser instanciado, embora sejam capazes de obter o resultado esperado, não possibilitam a evolução ou o surgimento de novos esquemas. Desta forma, sempre que a estrutura de um *Schema* é alterada, ou novos esquemas são exigidos, novas implementações no código fonte de tais sistemas são necessárias.

Outra forma de gerar documentos JSON é através de consultas *SQL* que, além de recuperar os dados, possuam fixados os nomes dos atributos previstos no *Schema*, e os *tokens* previstos para a formatação JSON. Estas consultas são capazes de gerar Documentos JSON, porém são especificas para cada *Schema*, e muito complexas quando se pretende utilizar esta

técnica para esquemas maiores. A Figura 4 demonstra a obtenção de um documento JSON por meio de *SQL* padrão (sem suporte nativo à JSON).

Figura 4 – Consulta *SQL* contendo fixos os nomes dos Atributos e formatação

```
with manager as
( select '{ '
       ||' "name":"'||ename||'"'
       ||',"salary":'||sal
       ||',"hiredate":"'||to_char(hiredate, 'DD-MM-YYYY')||'"'
       ||'} ' json
  , emp.*
  from   emp
)
, employee as
( select '{ '
       ||' "name":"'||ename||'"'
       ||',"job":"'||job||'"'
       ||',"salary":'||sal
       ||',"manager":'||case when mgr is null then '""'
                             else (select json
                                   from manager mgr
                                   where mgr.empno = emp.mgr)
                        end
       ||',"hiredate":"'||to_char(hiredate, 'DD-MM-YYYY')||'"'
       ||'} ' json
  , emp.*
  from   emp
)
, department as
( select '{ '
       ||' "name":"'||dname||'"'
       ||',"identifier":"'||deptno||'"'
       ||',"location":"'||loc||'"'
       ||',"employees":'||(select '['||listagg( json, ',')
                                       within group (order by 1)
                                  ||']' as data
                           from employee emp
                           where emp.deptno = dept.deptno
                          )
       ||'} ' json
  from   dept
)
select '{"company" : ['
       ||( select listagg( json, ',')
                  within group (order by 1)
           from   department)
       ||']}'
from   dual
```

Fonte: (JELLEMA, 2011)

Utilizando as mais recentes versões dos principais bancos de dados relacionais, a obtenção de documentos JSON é facilitada utilizando as funções preestabelecidas no padrão *SQL/JSON*. Desta forma, as consultas *SQL* recuperarão os dados e farão a serialização no padrão esperado, porém serão específicas para cada esquema, e a complexidade de implementá-las é diretamente proporcional a complexidade dos *Esquemas* JSON utilizados. A Figura 5

representa a utilização das funções *JSON_object* e *JSON_arrayagg* implementadas no Banco de Dados *Oracle*.

Figura 5 – Consulta *SQL* com funções nativas do *Oracle* para saída JSON

```
select json_object(
  'id' is department_id,
  'name' is department_name,
  'location' is (
    select json_object(
      'id' is location_id,
      'streetAddress' is street_address,
      'postalCode' is postal_code,
    )
    from locations loc
    where location_id = dept.location_id
  ),
  'manager' is (
    select json_object(
      'id' is employee_id,
      'name' is first_name || ' ' || last_name,
      'salary' is salary,
    )
    from employees man
    where employee_id = dept.manager_id
  ),
  'employees' is (
    select json_arrayagg(
      json_object(
        'id' is employee_id,
        'name' is first_name || ' ' || last_name,
        'jobHistory' is (
          select json_arrayagg(
            json_object(
              'departmentId' is department_id,
              'startDate' is to_char(start_date, 'DD-MON-YYYY'),
              'endDate' is to_char(end_date, 'DD-MON-YYYY')
            )
          )
          from job_history
          where employee_id = emp.employee_id
        )
      )
    )
    from employees emp
    where department_id = dept.department_id
  )
) as department
from departments dept
where department_id = :department_id
```

Fonte: (MCGHAN, 2018)

Nos exemplos das Figuras 4 e 5 é possível observar que em ambas foram utilizadas consultas aninhadas para a recuperação dos dados. Na Figura 4 os elementos de formatação

para a apresentação dos dados foram fixados na consulta; já na Figura 5 foram utilizadas funções nativas que se encarregam de serializar os dados no padrão de saída JSON.

2.3.1 Trabalhos Relacionados

Dentre as diversas fontes pesquisadas, os trabalhos que melhor se relacionam com a proposta apresentada nesta dissertação são os que tratam da problemática da migração entre bancos de dados. A migração envolvendo os paradigmas relacional e orientado a documentos envolvem desafios semelhantes aos encontrados durante o desenvolvimento deste trabalho. O trabalho de KARNITIS et al. (2015) define uma ferramenta denominada *DigiBrowser* para navegação entre os relacionamentos de bases de dados relacionais e a migração para bancos de dados *NoSQL*. A ferramenta proposta possibilita a conversão entre bancos de dados relacionais como *Oracle, PostgreSQL e MySQL* para os formatos *XML* e *JSON*. Para isso, divide o processamento em duas fases, inicialmente o esquema do banco de dados relacional e metadados como tabelas, colunas, chaves e visões são obtidos, em seguida procede com a conversão para o modelo de documentos. Em um dado momento, anterior a conversão, é requisitado o auxílio do usuário para a classificação das tabelas nos seguintes tipos: "Codificadas" (tabelas que contêm dados codificados como tabela de cores contendo o campo identificador e descrição das cores); "Entidade Simples" (tabelas que contém todos os atributos de determinado domínio); "Entidade complexa" (tabelas que contenham dados fragmentados em outras tabelas); e "Relacionamento N:N" (tabela intermediárias que resolvem relacionamentos N:N).

Após a classificação das tabelas do esquema de dados, um algoritmo proposto analisa as relações entre as tabelas por meio da associação de chaves estrangeiras. Esta análise limita-se a um único nível de relacionamento, gerando um modelo que servirá de base para a análise do usuário que manualmente realiza os ajustes necessários. Este trabalho exige a intervenção do usuário tanto para classificar as tabelas quanto para analisar e ajustar o esquema de dados proposto. Possui semelhança com a proposta deste trabalho por converter dados originados do modelo relacional para o modelo semiestruturado. A diferença é que neste trabalho o processo automatiza a obtenção do modelo resultante, por meio da análise automatizada de esquemas *JSON*, e da utilização de consultas *SQL* para a obtenção dos dados. Diferentemente da proposta de KARNITIS et al., a solução proposta neste trabalho não possui o objetivo de converter uma base de dados inteira para o modelo de documentos. O resultado desejado neste trabalho é gerar

documentos na notação JSON de forma automatizada, mantendo o escopo semântico definido nos esquemas JSON preconcebidos.

A proposta de ZHAO et al., (2014) prevê o armazenamento das colunas das tabelas referenciadas agregando-as ao item de dados da tabela que as referencia. A conversão entre os modelos relacional e de documentos depende da extensão dos dados podendo ser:

- Simples (tabela relacional que referencia uma ou nenhuma outra tabela);

- Vertical (relacionamento em profundidade) - Quando uma tabela referencia outra que por sua vez referencia uma terceira e assim sucessivamente.

  Exemplo: se a tabela “T1” referencia “T2”, a estrutura de dados resultante será obtida pela agregação dos atributos da tabela “T2” na tabela “T1”, na ocorrência de vários níveis de relacionamento em profundidade, as demais tabelas são agregadas de forma sequencial.

- Horizontal (relacionamentos não encadeados)

  Exemplo: se a tabela “T1” referencia as tabelas T2, T3 e T4, a conversão resultará em um documento “T1” que possuirá os itens de “T2”, “T31” e “T4”.

O trabalho de ZHAO et al., executa as conversões entre os modelos de dados iniciando pelas extremidades, ou seja, no sentido de fora para dentro. Fazendo uso de recursividade as tabelas que não referenciam nenhuma outra são as primeiras a serem convertidas e assim sucessivamente. Após executar a migração por meio de um protótipo desenvolvido, experimentos utilizando consultas equivalentes em ambas as abordagens demonstraram a possibilidade de se obter os mesmos dados provindos do banco relacional original e do banco orientado a documentos resultante.

Ao comparar a proposta desta dissertação, ao trabalho de ZHAO et al., observa-se a semelhança quanto ao uso de recursividade para a conversão dos dados relacionais em dados semiestruturados. A diferença é que na proposta de ZHAO et al., os documentos são gerados agregando sequencialmente os dados das tabelas referenciadas nas tabelas que as referencia; desta forma os documentos obtidos são agregações de tabelas, sem a preocupação de manter a lógica semântica no modelo de documentos semiestruturados resultante. A proposta desta

dissertação obtém os documentos por meio da execução sequencial de consultas associadas aos nós da árvore hierárquica dos esquemas JSON. O processamento ocorre no sentido do nó raiz para as pontas, ou seja, do centro para as extremidades. Os resultados obtidos são documentos instanciados a partir de um esquema JSON automaticamente mapeado, mantendo assim a semântica das informações conforme o escopo do esquema.

2.4 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Como o objetivo deste trabalho é a geração automatizada de documentos *JSON*, por meio do mapeamento dos seus esquemas, e da obtenção de dados originados do modelo relacional, é necessário compreender os conceitos que envolvem os paradigmas de armazenamento relacional e semiestruturado. O modelo relacional é fortemente atrelado a um esquema predefinido, este esquema garante a integridade dos dados, porém, por exigir uma estruturação rígida e inflexível, lança desafios ao se deparam com fontes de dados heterogêneas. O modelo semiestruturado, embora possa possuir esquema que o defina, este é de fácil adaptação à ocorrência de mudanças no escopo dos dados a serem armazenados. O modelo semiestruturado é o modelo mais utilizado para o intercâmbio de informações entre diferentes sistemas e para a publicação de dados na web. Dentre os padrões de arquivos do modelo semiestruturada, os mais representativos são XML, YAML e JSON. O padrão XML, por muitos anos, fora considerado como preferencial para a transmissão de dados, porém, nos últimos anos, o padrão *JSON* vem ganhando destaque e tende a gradativamente substituir o XML.

Compreender o processo de normalização, conceito fundamental para a estruturação do modelo relacional, bem como compreender a serialização dos dados no modelo semiestruturado, é essencial para a conversão entre os modelos e assim obter documentos semiestruturados a partir do modelo relacional. Neste trabalho, o processo de geração é guiado por metadados obtidos através do mapeamento de esquemas JSON, sendo necessário conhecer as principais propriedades que os definem.

Dentre os trabalhos analisados, os que mais se aproximam da proposta apresentada nesta dissertação, são os estudos sobre a migração de bancos relacionais para orientados a documentos. Estes procuram converter bancos de dados inteiros, por meio de processos de agregação de tabelas através do relacionamento encadeado de suas chaves estrangeiras. Basicamente agregam as tabelas relacionadas incorporando seus campos na tabela que as referenciam. Já a proposta descrita nesta dissertação permite a definição do escopo e da estrutura dos dados. Neste trabalho o processo de geração dos documentos json é guiado por

esquemas mapeados. A principal vantagem deste processo é a obtenção de documentos em conformidade com a semântica implícita nos esquemas. Outra vantagem é possibilidade de utilizar o processo para múltiplos propósitos, com: A obtenção de documentos para compor coleções em um banco de dados NoSQL orientado a documentos; a geração de documentos para a interoperabilidade entre sistemas distintos; a obtenção de dados para a publicação em portais da internet; entre outros.

3 UM PROCESSO PARA A GERAÇÃO AUTOMATIZADA DE INSTÂNCIAS DE ESQUEMAS JSON A PARTIR DE CONSULTAS SQL

Este capítulo descreve o processo proposto para a geração de documentos semiestruturados na notação JSON, instâncias de esquemas JSON previamente informados. Tendo como origem dados armazenados no modelo relacional, a execução em cadeia de consultas *SQL* recupera os dados que são anotados em conformidade com os metadados mapeados do esquema. O processo inicialmente recebe a entrada de um esquema JSON, este esquema é automaticamente mapeado e seus metadados armazenados em tabelas especificas do modelo relacional. O mapeamento do esquema distingue propriedades estruturais de propriedades simples. As propriedades estruturais são tratadas como nós da árvore hierárquica prevista no esquema, a elas são associadas consultas *SQL* que recuperarão os dados das tabelas relacionais. Propriedades simples são elementos não estruturais, sendo estes, atributos pertencentes a um determinado nó. A cada um destes atributos é associado um campo da projeção da consulta correspondente ao nó do qual o atributo pertence. O processamento de algoritmos recursivos executará sequencialmente as consultas *SQL* na ordem exata dos nós da árvore do esquema. Durante a execução das consultas os documentos *JSON* são gerados, constituindo a saída esperada para o processo proposto.

Este capítulo descreve a carga dos esquemas JSON e a resolução das referências que poderão ser encontradas em suas propriedades, tais referências apontam para as definições detalhadas destas propriedades. Após a resolução das referências é executada a análise do esquema, e realizada a coleta e o armazenamento de seus objetos, vetores e propriedades simples. Os artefatos analisados passarão por um pré-processamento que visa manter a hierarquia e sequência dos objetos, vetores e demais propriedades. Estes elementos, na etapa de processamento, serão percorridos na sequência exata em que estão dispostos no esquema carregado. Durante o processamento, objetos e vetores serão interpretados como nós da árvore hierárquica prevista no esquema. A cada um dos nós uma consulta *SQL* será executada recuperando os dados que serão inseridos no documento JSON a ser gerado.

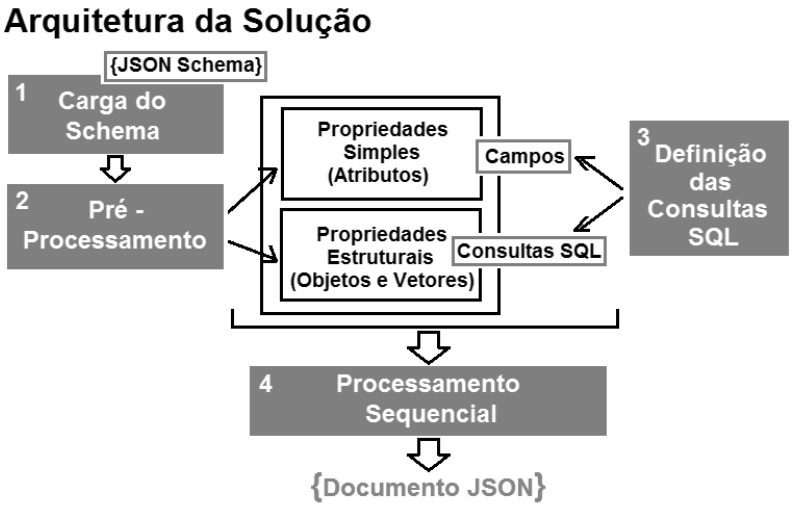
As técnicas descritas neste capítulo possuem como premissa a compatibilidade com a maioria dos bancos de dados relacionais, pois as instruções utilizadas fazem parte da sintaxe de consulta *SQL* padrão. A compatibilidade com diferentes esquemas JSON também é garantida, pois o processamento de tais esquemas será realizado através de algoritmos recursivos, estes possuem a capacidade de processar a estrutura dos esquemas independentemente da quantidade de níveis que apresentarem.

Para melhor descrever algumas técnicas, estas são apresentadas usando BPMN (*Business Process Model and Notation)* (BROCKE e ROSERMANN, 2010). Desta forma, utilizando a diagramação apropriada destes processos, a compreensão dos detalhes de tais técnicas é facilitada. O capítulo 4 descreve algoritmos detalhados para as etapas do processo apresentado neste capítulo.

3.1 VISÃO GERAL DA PROPOSTA

A estruturação de um documento JSON frequentemente é definida por meio de um esquema constituído por tipos primários de dados, tais como: objetos literais, matrizes, cadeias de caracteres, numerais, booleanos e nulo. O processo proposto prevê inicialmente a carga e análise automática destes esquemas, de modo que seus metadados sejam armazenados em tabelas relacionais para posterior processamento. Após o mapeamento dos esquemas são definidas as consultas *SQL*, estas são responsáveis pela recuperação dos dados necessários a geração dos documentos JSON instanciados a partir destes esquemas. Os documentos gerados resultarão da execução automática e sequencial de tais consultas. A medida que a execução avança, são anotados os pares chave-valor na estrutura exata definida pelo esquema previamente carregado. A Figura 6 ilustra os aspectos gerais da arquitetura proposta.

Figura 6 - Arquitetura da solução proposta



Fonte: Autor.

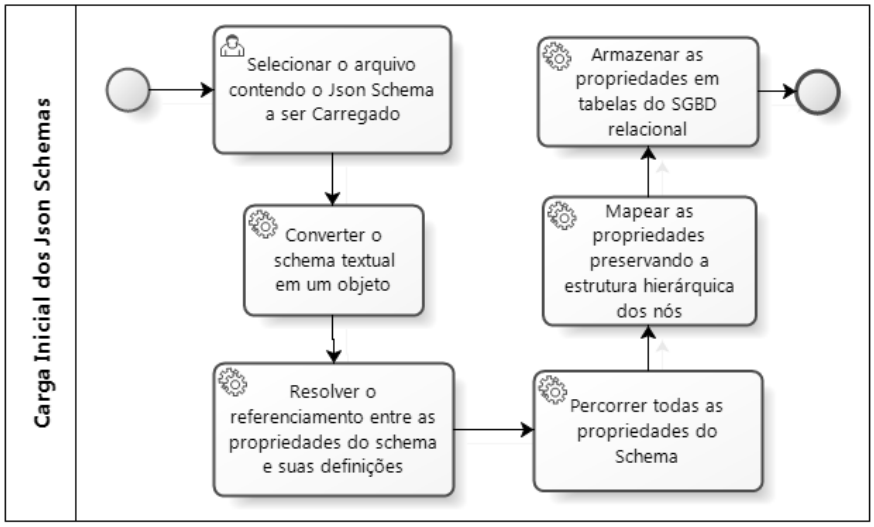
Inicialmente é realizada a carga do esquema. Nesta etapa deverá ser selecionado o arquivo que contém o esquema a ser carregado. A etapa de pré-processamento objetiva

distinguir as propriedades anotadas no esquema, classificando-as em propriedades estruturais e propriedades simples. Tais propriedades e suas características constituem os metadados do esquema. Na etapa de definição das consultas *SQL*, cada um dos nós, propriedades estruturais do esquema, receberá uma consulta responsável por recuperar os dados correspondentes aos valores de seus atributos (propriedades simples). As associações dos campos da projeção das consultas definidas para os nós, com seus respectivos atributos, formarão os pares chave-valor do documento a ser gerado. Os detalhes das etapas, representadas na Figura 6, estão descritos nas subseções seguintes.

3.2 CARGA DOS ESQUEMAS

Os esquemas JSON definem a estrutura e o escopo dos dados que os documentos *JSON* devem possuir. Assim como os documentos JSON instanciados a partir de determinado esquema, o próprio esquema JSON é um documento JSON válido. O processo de carga do esquema JSON, inicialmente deverá promover a resolução das referências que poderão ser encontradas em suas propriedades. Tais referências apontam para as definições detalhadas destas propriedades. Após a resolução das referências, as propriedades e suas especificações deverão ser mapeadas para tabelas relacionais. O diagrama da Figura 7 ilustra os passos a serem seguidos para a carga dos esquemas.

Figura 7 - Visão geral da carga dos *Esquemas* JSON



Fonte: Autor.

3.2.1 Carga Inicial

A solução proposta requer que os objetos JSON que compõem o esquema sejam navegáveis. Desta forma é possível a manipulação dos mesmos para a coleta e análise de suas características. Para a solução deste problema foram implementados neste trabalho métodos que permitem o tratamento destas referências. Somente após este tratamento fora possível a carga completa do esquema e a navegabilidade entre os objetos JSON obtidos.

3.2.2 Resolução de Referências

Para resolver as referências foram desenvolvidos algoritmos que permitem o tratamento destas. Os algoritmos desenvolvidos com o objetivo de resolver as referências encontram-se na seção 4.1. A Figura 8 ilustra um trecho de esquema que utiliza propriedades referenciadas.

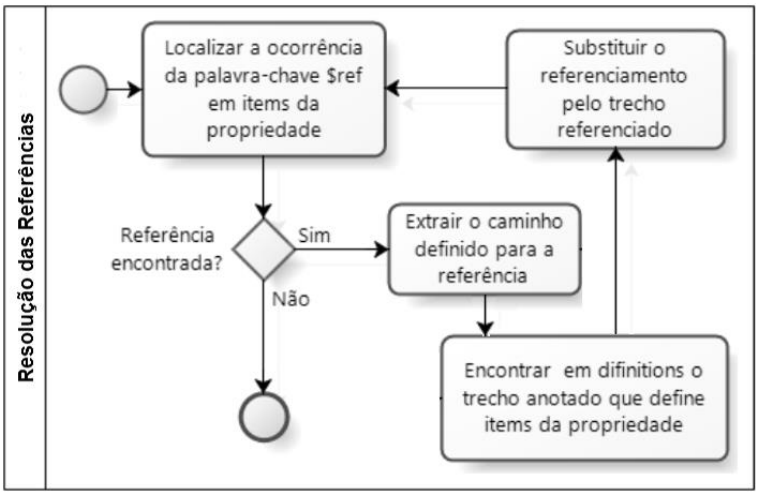
Figura 8 - Trecho de um *Esquema* JSON contendo referenciamentos

Fonte: Autor.

No exemplo acima a propriedade *lote* (linha 31), do tipo *array*, determina através da palavra-chave *$ref* (linha 34) o caminho para a definição de seus itens. Observa-se que o caminho definido em *$ref* aponta para *definitions/lote*, permitindo assim a localização da definição de *lote* (linha 3). Ainda é possível observar que *lote* possui outra propriedade *array* denominada *item*, esta por sua vez aponta para *definitions/item*, cuja definição encontra-se em *item* (linha 18).

A análise de todas as características das propriedades encontradas em um esquema *JSON* é facilitada após a resolução das referências. A Figura 9 representa a identificação e resolução das referências.

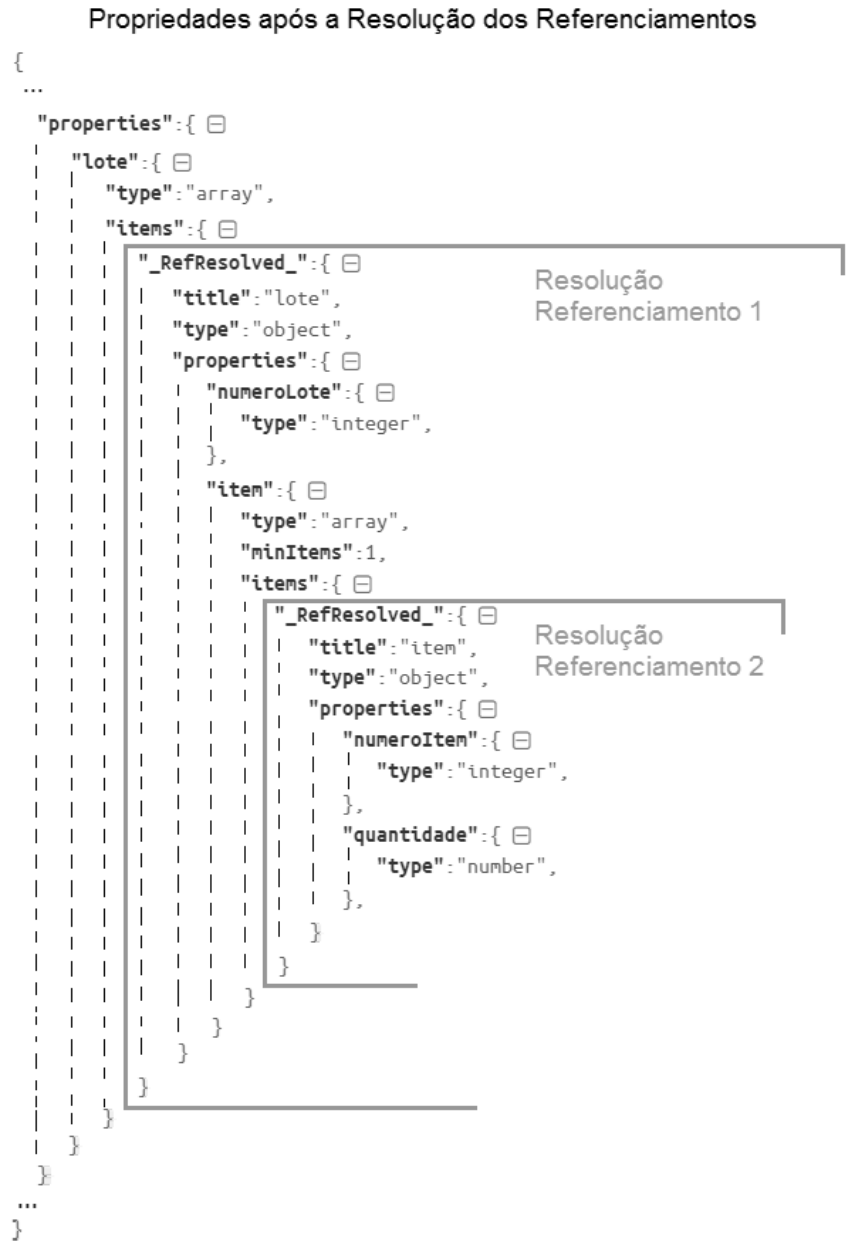
Figura 9 - Diagrama de resolução das referências



Fonte: Autor.

Para a resolução das referências, as propriedades são percorridas em busca da palavra-chave *$ref* associada a *items*. Após a localização do referenciamento, o caminho (valor de *$ref*) é identificado, possibilitando a localização do trecho de anotação correspondente. Desta forma todas as referências são resolvidas sequencialmente, através da sobreposição destas ao conjunto chave-valor definido pela palavra-chave *$ref* e seu respectivo valor (caminho). A Figura 10 ilustra o esquema após a resolução das referências.

Figura 10 - Trecho de Documento JSON após resolução dos referenciamentos



Fonte: Autor.

Ao término desta etapa, a saída obtida é o mesmo esquema JSON, porém em uma versão cujas propriedades possuem itens definidos na própria anotação da propriedade, portanto sem a utilização de referenciamento. Após a resolução das referências é executado o pré-processamento dos esquemas, este procedimento consiste na análise do esquema JSON, coleta e armazenamento de seus objetos, matrizes e propriedades simples.

3.3 PRÉ-PROCESSAMENTO

A análise do esquema, após a resolução das referências encontradas, tem o objetivo de classificar as propriedades em propriedades estruturais ou simples. As propriedades estruturais são objetos e matrizes que formarão a estrutura dos documentos JSON a serem instanciados. Propriedades simples definem as chaves que formarão o conjunto chave-valor responsável por armazenar os dados associados às propriedades estruturais. A distinção das propriedades em simples e estruturais, é necessária para que seja possível o processamento e obtenção dos resultados esperados neste trabalho.

3.3.1 Análise e armazenamento do Esquema

A análise do esquema após a resolução das referências encontradas tem o objetivo de classificar as propriedades em propriedades estruturais e propriedades simples. A Figura 11 a seguir é um extrato de um esquema JSON, neste é possível observar exemplos de propriedades estruturais e simples.

Figura 11 – Exemplo de propriedades de um *Esquema* JSON

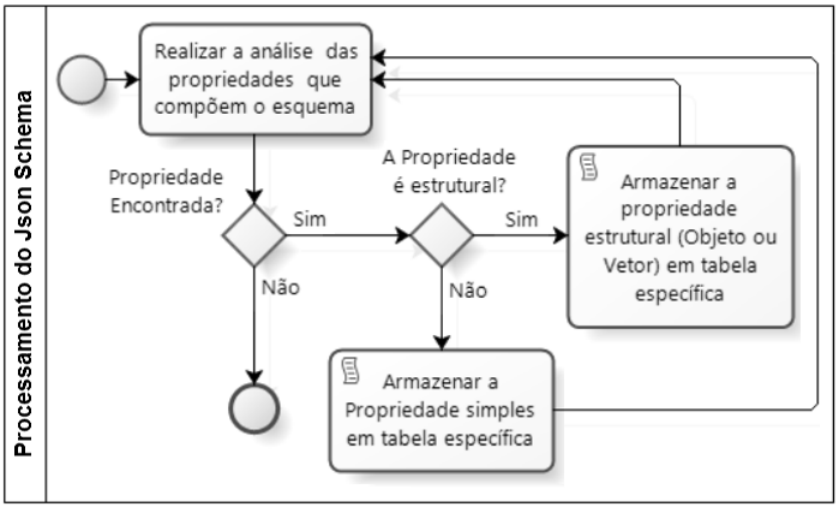
```
{
    "$schema": "http://teste.br/schema#",
    "title": "Processolicitacao",
    "description": "Dados iniciais do processo de licitacao",
    "type": "object",
    "definitions": {
        "lote": {
            "title": "lote",
            "type": "object",
            "properties": {
                "numeroLote": {
                    "type": "integer",
                },
                "descricaoLote": {
                    "type": "string",
                },
                "item": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/item"
                    }
                }
            }
        },
        "item": {
            "title": "item",
            "type": "object",
            "properties": {
                "numeroItem": {
                    "type": "integer",
                },
                "quantidade": {
                    "type": "number",
                }
            }

        }
    },
    "properties": {
        "exercicioLicitacao": {
            "type": "integer",
        },
        "dataEdital": {
            "type": "string",
            "format": "date"
        },
        "lote": {
            "type": "array",
            "items": {
                "$ref": "#/definitions/lote"
            }
        }
    }
}
```

Fonte: Autor.

No trecho de esquema representado acima é possível observar as propriedades simples *exercicioLicitacao, dataEdital, numeroLote, descricaoLote, numeroItem* e *quantidade*, algumas destas associadas à raiz do esquema, outras associadas a uma das propriedades estruturais *lote* e *item*.

O diagrama da Figura 12, a seguir, representa o processamento de um esquema. Este analisa cada propriedade encontrada segregando-as em simples ou estruturais. Durante este processo, as propriedades, dependendo de seu tipo, são armazenadas em tabelas relacionais distintas.

Figura 12 - Processamento e armazenamento do esquema *JSON*

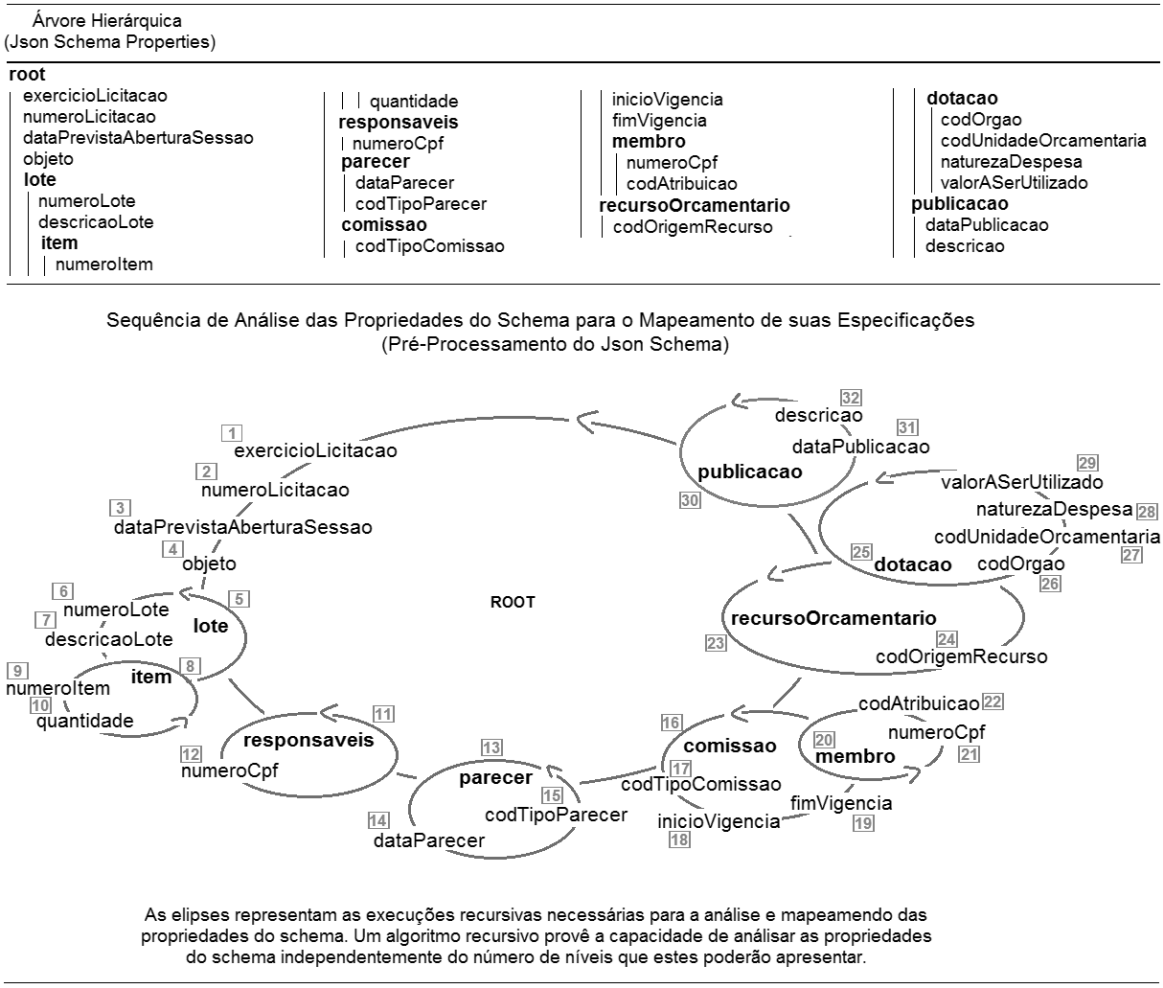


Fonte: Autor.

A etapa de pré-processamento e análise dos esquemas é realizada por meio de um algoritmo recursivo, este possui a capacidade de percorrer os nós independentemente do número de níveis e complexidade estrutural dos esquemas. O algoritmo que processa o mapeamento dos esquemas é descrito detalhadamente na seção 4.2.

A Figura 13 exemplifica a sequência de execução da etapa de pré-processamento. A sequência de propriedades do esquema JSON é percorrida, iniciando pela propriedade estrutural raiz, todas as propriedades são analisadas, independentemente do nível de profundidade em que se encontram. Sempre que uma propriedade estrutural é encontrada, uma chamada recursiva é realizada para mapear seus elementos. Os elementos de uma propriedade estrutural poderão ser propriedades simples ou outras propriedades estruturais filhas desta.

Figura 13 – Sequência de execução da etapa de pré-processamento



Fonte: Autor.

A execução do algoritmo inicia pelo processamento da propriedade raiz, seguindo para as extremidades, analisando e mapeando para tabelas relacionais todas as propriedades e suas caracteristicas fundamentais. Para possibilitar a execução das etapas posteriores, é necessário obter as caracteristicas de cada propriedade. As caracteristicas essenciais para a estruturação dos documentos, a serem gerados ao término do processo proposto neste trabalho, são:

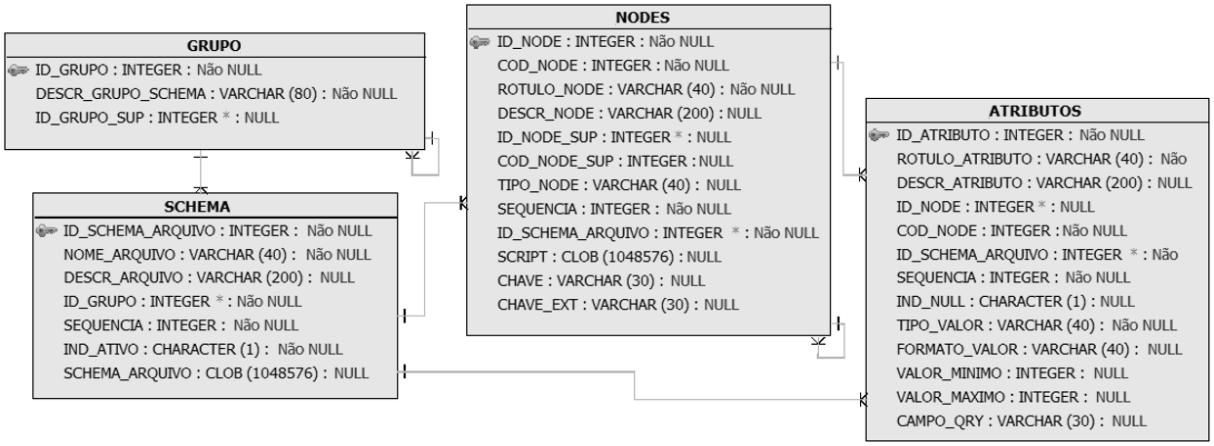
- Para propriedades estruturais: itentificador do nó, identificador do nó superior, e rótulo;

- Para propriedades simples: identificador da propriedade, rótulo, posição, tipo, e identificador do nó que a contém.

3.3.1.1 Modelo Relacional Proposto para o Armazenamento dos Esquema Mapeados

De forma a armazenar as propriedades do esquema, foram implementadas tabelas relacionais. O armazenamento dos metadados do esquema é necessário para posterior processamento da geração dos documentois JSON. O armazenamento do esquema e suas propriedades está representado na Figura 14.

Figura 14 – Modelo proposto para a persistência do esquema *JSON* mapeado



Fonte: Autor.

No modelo proposto, a tabela SCHEMA é responsável pelo armazenamento do esquema na notação JSON (coluna SCHEMA_ARQUIVO), já as tabelas NODES e ATRIBUTOS destinam-se respectivamente ao armazenamento das propriedades estruturais e propriedades simples do esquema carregado. A tabela GRUPO fora incluída para melhor organização dos esquemas, e não é utilizada para o processamento dos documentos a serem gerados.

O campo SCRIPT da tabela NODES definirá a Instrução SQL a ser executada para a obtenção dos dados que "alimentarão" as propriedades relacionadas ao nó (*node*) correspondente. A estrutura hierárquica dos nós é preservada por meio de auto relacionamento através das colunas ID_NODE e ID_NODE_SUP. Ainda nesta tabela a coluna CHAVE_EXT armazenará o campo, da projeção da consulta, que servirá de chave para os nós-filhos (parâmetro a ser repassado às consultas dos nós descendentes). O campo SEQUENCIA define a ordenação de processamento dos nós, esta ordenação é idêntica a sequência com que as propriedades estruturais aparecem no esquema. A tabela ATRIBUTOS armazenará todas as propriedades simples e suas características. Nesta tabela, o campo ROTULO_ATRIBUTO definirá a chave do conjunto chave-valor para as propriedades do documento JSON a ser

gerado. Já a coluna CAMPO_QRY armazenará um dos campos da projeção da consulta SQL do nó associado a propriedade, este campo originará o valor do conjunto chave-valor da propriedade no documento JSON. Também em ATRIBUTOS o campo SEQUENCIA preservará a ordenação que deverá ser respeitada ao inserir as propriedades no documento resultante.

Observa-se que o campo ID_NODE, chave estrangeira (FK) entre as tabelas NODES e ATRIBUTOS, não foi definido, isto deve-se à característica desta proposta que insere as propriedades em ATRIBUTOS antes da inserção dos nós em NODES. O procedimento para alimentar ID_NODE será detalhado posteriormente neste trabalho.

Após a execução do mapeamento realizado na etapa de pré-processamento, as tabelas relacionais, denominadas *nodes* e *attributes*, para o exemplo acima, são alimentadas conforme representado no quadro 6 a seguir:

Quadro 6 – Disposição dos metadados do *Esquema JSON* no banco relacionais

Properties						
Estruturais (Nodes)				**Simples (Attributes)**		
Rótulo Node	**type**	**Node Superior**	**Level**	**Rótulo Attributes**	**type**	**format**
root	object		1	exercicioLicitacao numeroLicitacao dataPrevistaAberturaSessao objeto	integer integer string string	 date
lote	array	root	2	numeroLote descricaoLote	integer string	
item	array	lote	3	numeroItem quantidade	integer number	
responsaveis	array	root	2	numeroCpf	string	
parecer	array	root	2	dataParecer codTipoParecer	string integer	date
comissao	array	root	2	codTipoComissao inicioVigencia fimVigencia	integer string string	 date date
membro	array	comissao	3	numeroCpf codAtribuicao	string integer	
recursoOrcamentario	array	root	2	codOrigemRecurso	integer	
dotacao	array	recursoOrcamentario	3	codOrgao codUnidadeOrcamentaria naturezaDespesa valorASerUtilizado	integer integer integer number	
publicacao	array	root	2	dataPublicacao descricao	string string	date

Fonte: Autor.

3.4 DEFINIÇÃO DAS CONSULTAS SQL

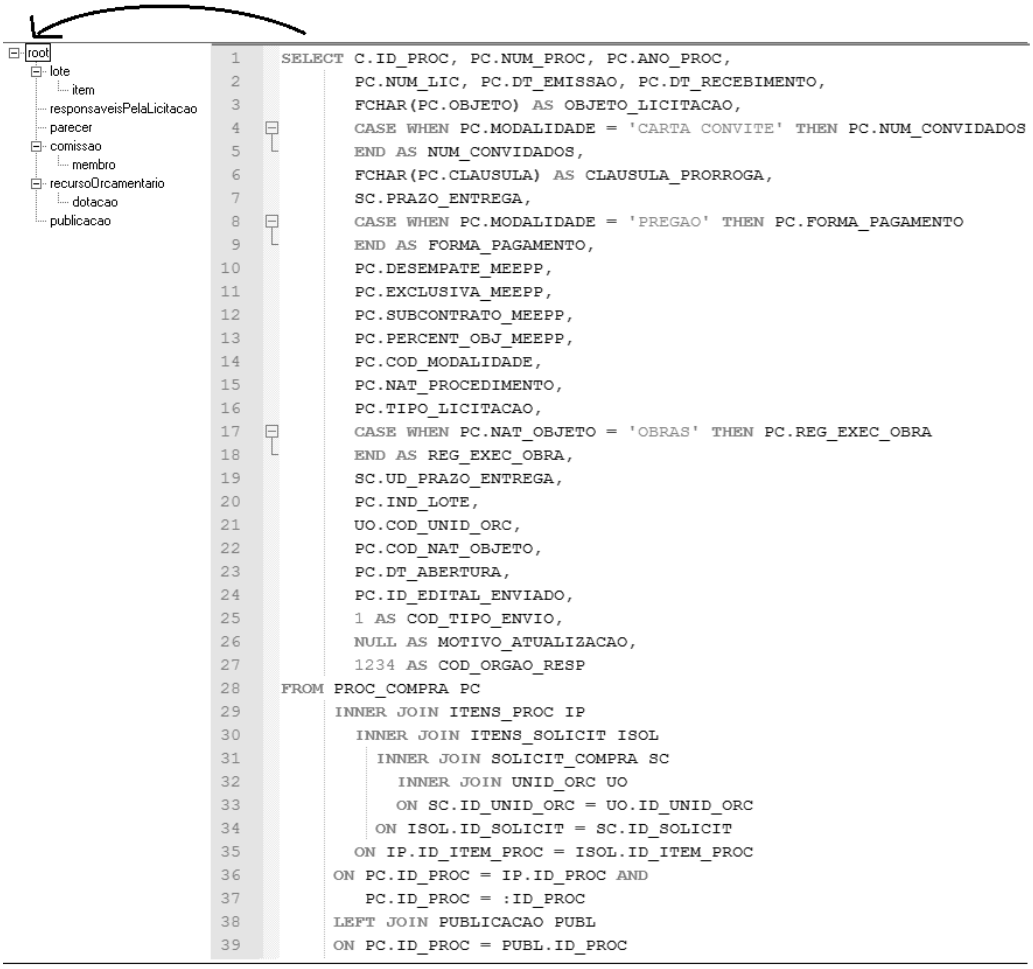
Nesta etapa ocorre interação com o usuário, onde este deverá informar as consultas *SQL* adequadas a cada um dos nós; bem como, associar aos atributos os respectivos campos da projeção das consultas correspondentes aos valores de cada um destes.

Os nós deverão receber as consultas *SQL* apropriadas para a obtenção dos valores para os seus atributos. Após a definição das consultas, deverão ser associados os atributos do nó aos campos da projeção da consulta correspondente. Ainda por meio da uma interface com o usuário, o mesmo deverá informar para cada nó que contenha descendentes, o campo de vinculação entre nó-pai e nó-filho. Estes campos chave entre os nós permitirão a execução sequencial das consultas *SQL* durante a etapa de geração dos documentos JSON correspondentes ao esquema. A Figura 15, descrita na seção seguinte, exemplifica as consultas associadas aos nós de determinado esquema. A interface com o usuário se encontra detelhada no Anexo 1 deste trabalho.

3.4.1 Cadastro de Consultas SQL

Os nós receberão um a um as consultas SQL necessárias à recuperação dos valores para os seus atributos. A Figura 15 demonstra a árvore de nós do esquema carregado e a consulta associada ao nó raiz.

Figura 15 – Árvores hierárquica de nós e a consulta *SQL* associada ao nó raiz



Fonte: Autor.

No exemplo ilustrado acima, a consulta apresentada corresponde ao nó raiz de determinado esquema JSON, sendo os campos projetados na consulta equivalentes aos atributos deste nó.

O Quadro 7 apresenta o resultado deste processo de inserção das consultas *SQL*, armazenadas na coluna SCRIPT da tabela NODES.

Quadro 7 – Representação da persistência das SQL Queries associadas aos nós

SCHEMA_NODES

ID_NODE	COD_NODE	ROTULO_NODE	ID_NODE_SUP	COD_NODE_SUP	TIPO_NODE	SEQUENCIA	SCRIPT
173	1	root	(null)	(null)	object	1	**SELECT ...**
175	2	lote	173	1	array	2	**SELECT ...**
174	3	item	175	2	array	3	**SELECT ...**
176	4	responsaveisPelaLicitacao	173	1	array	4	**SELECT ...**
177	5	parecer	173	1	array	5	**SELECT ...**
179	6	comissao	173	1	array	6	**SELECT ...**
178	7	membro	179	6	array	7	**SELECT ...**
181	8	recursoOrcamentario	173	1	array	8	**SELECT ...**
180	9	dotacao	181	8	array	9	**SELECT ...**
182	10	publicacao	173	1	array	10	**SELECT ...**

Fonte: Autor.

No exemplo acima é possível observar que o campo SCRIPT da tabela NODES armazena as consultas informadas pelo usuário. As consultas deverão ser implementadas utilizando *SQL* padrão, resultando em dados tabulares que automaticamente serão convertidos em formato JSON.

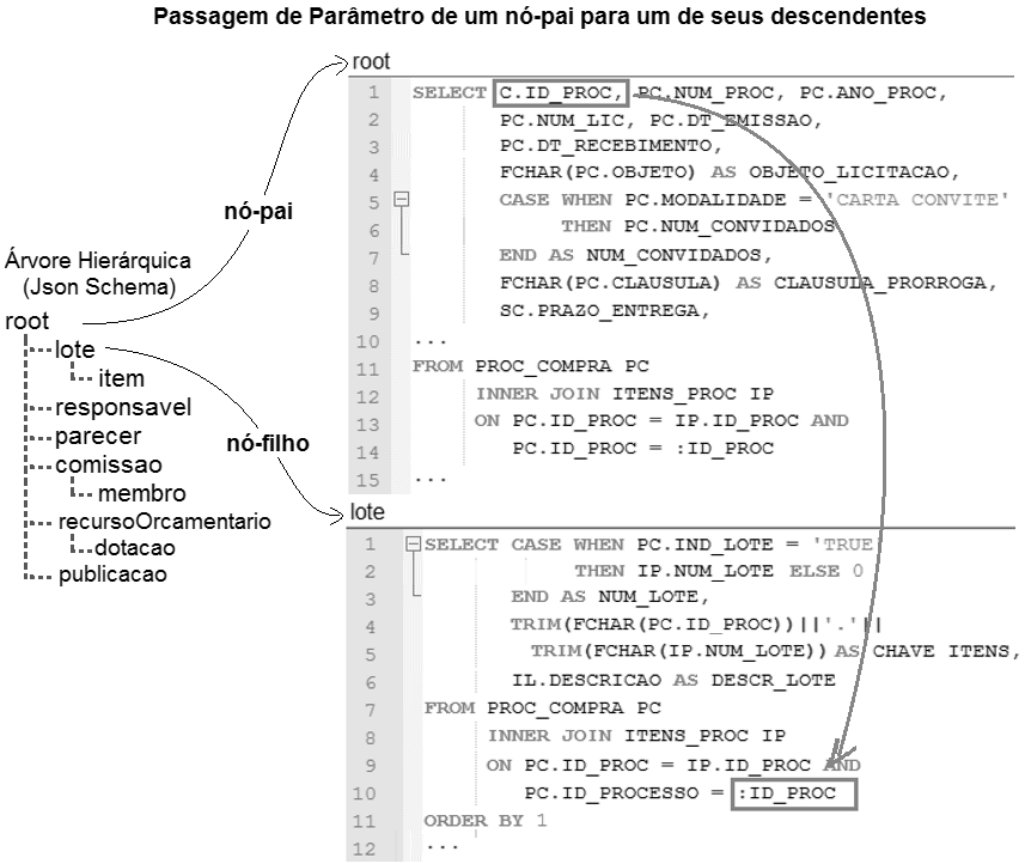
Para cada nó do esquema (propriedade estrutural) é necessário associar uma consulta capaz de obter os dados para seus atributos (propriedades simples). Os campos da projeção das consultas proveem os valores para os atributos previstos no esquema. As consultas implementadas devem atender ao escopo determinado pelo esquema mapeado. Esquemas complexos exigirão consultas igualmente complexas. Tais consultas muitas vezes envolvem junções entre diversas tabelas relacionais e clausulas específicas que dependerão do domínio dos dados a serem retornados. A solução proposta automatiza a análise dos esquemas e a estruturação dos dados no padrão JSON. A independência de esquemas específicos permite obter documentos JSON para múltiplos propósitos. Independentemente da complexidade, qualquer esquema JSON válido poderá ser processado.

3.4.2 Configuração dos Parâmetros para repasse entre nós-pai e dependentes

A definição de uma coluna da projeção da consulta do nó-pai para que seja repassada aos seus descendentes possibilita automatizar a execução recursiva sequencial. O processamento para a obtenção dos documentos semiestruturados iniciará pelo nó raiz e automaticamente percorrerá todos os demais nós executando suas respectivas consultas.

O exemplo da Figura 16, a seguir, representa as consultas relacionadas ao nó-pai denominado raiz, e um de seus nós-filhos de rótulo lote.

Figura 16 - Representação da passagem de parâmetro entre nó-pai e nó-filho



Fonte: Autor.

Na representação acima, o campo *ID_PROC* projetado na consulta associada ao nó-pai raiz, fora definido como chave, parâmetro, para a consulta relacionada com o nó-filho lote. Sendo assim, após a execução da consulta do nó raiz, é possível "disparar" a consulta do nó lote. O mesmo ocorrerá com todos os nós hierárquicos do esquema. O quadro 8 destaca o armazenamento dos campos chaves entre nós-pai e filhos.

Quadro 8 – Persistência dos campos chaves de interligação entre nó-pai e filhos

SCHEMA_NODES

ID_NODE	COD_NODE	ROTULO_NODE	ID_NODE_SUP	TIPO_NODE	SEQUENCIA	SCRIPT	CHAVE_EXT
173	1	root	(null)	object	1	SELECT ...	**ID_PROC**
175	2	lote	173	array	2	SELECT ...	**CHAVE_ITENS**
174	3	item	175	array	3	SELECT ...	(null)
176	4	responsaveisPelaLicitacao	173	array	4	SELECT ...	(null)
177	5	parecer	173	array	5	SELECT ...	(null)
179	6	comissao	173	array	6	SELECT ...	**ID_COMISSAO**
178	7	membro	179	array	7	SELECT ...	(null)
181	8	recursoOrcamentario	173	array	8	SELECT ...	**ID_FONTE_REC**
180	9	dotacao	181	array	9	SELECT ...	(null)
182	10	publicacao	173	array	10	SELECT ...	(null)

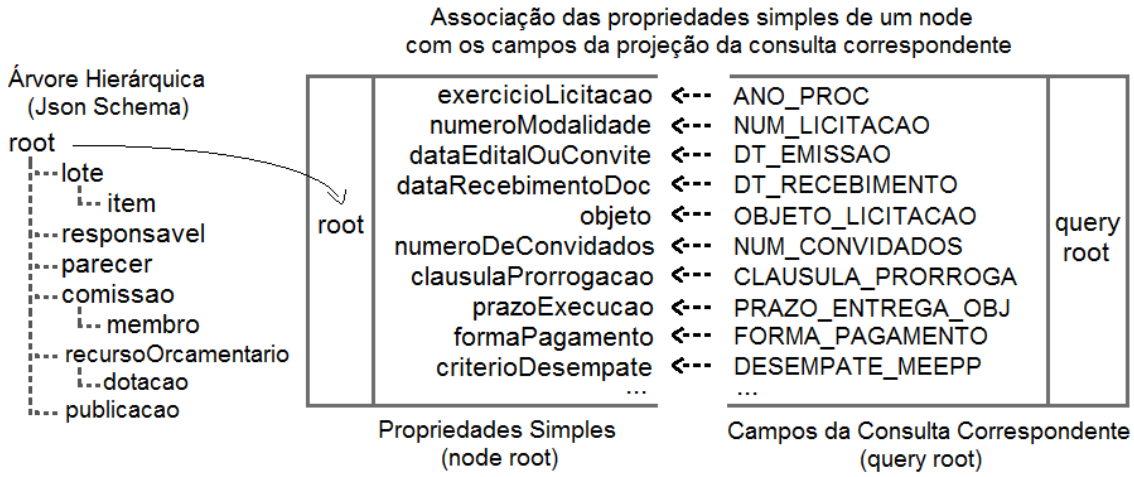
Fonte: Autor.

Observa-se que somente os nós-pais terão a coluna CHAVE_EXT da tabela NODES “alimentada”, esta coluna define o campo de origem do valor a ser repassado para os nós-filhos durante o processamento da geração dos documentos JSON correspondentes.

3.4.3 Associação dos Campos da Consultas e as Propriedades do Esquema

Para a geração dos documentos JSON é necessário obter os conjuntos chave-valor das informações que os compõem. Tais elementos são obtidos através da vinculação entre as propriedades simples, atributos dos nós estruturais, e os campos da projeção da consulta correspondente ao dado desejado. O Quadro 9 ilustra a vinculação necessária entre atributos do nó raiz e respectivos campos da consulta.

Quadro 9 – Associação dos campos da consulta às propriedades dos esquemas



Fonte: Autor.

Para cada nó deverá existir uma consulta *SQL* associada, sendo que para cada atributo deste nó deverá ser vinculada uma coluna da projeção desta consultada.

A associação entre atributos e os campos da projeção das consultas se faz por meio de uma interface implementada para interação com o usuário. A vinculação entre atributos e campos resulta no armazenamento de tais informações na coluna CAMPO_QRY da tabela ATRIBUTOS. O Quadro 10 representa os campos da projeção da consulta informada para o nó raiz, vinculados aos atributos deste mesmo nó.

Quadro 10 – Persistência da associação entre campos da consulta SQL e atributos dos nós

NODE_ATRIBUTOS

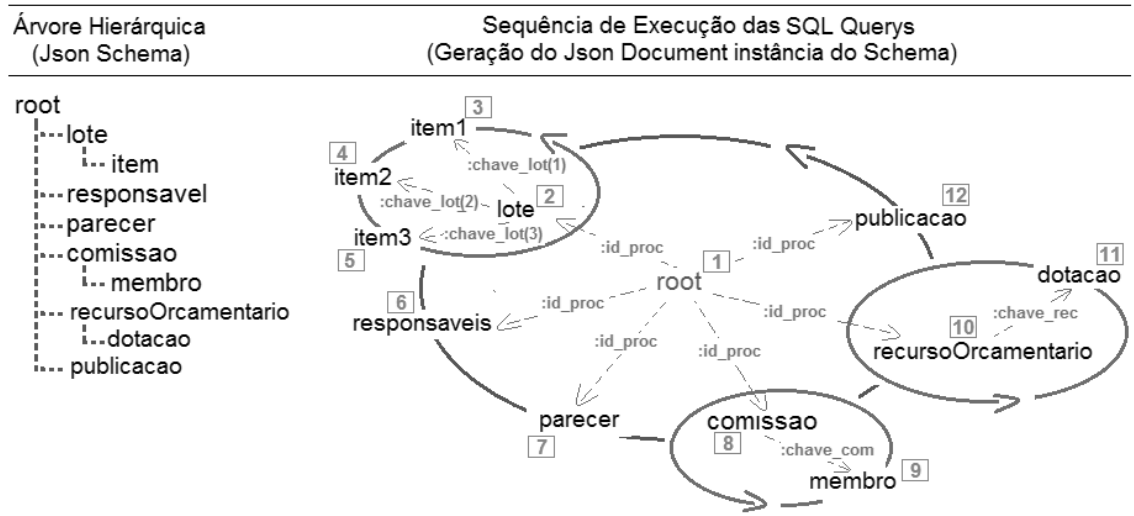
ID_ATRIBUTO	ROTULO_ATRIBUTO	ID_NODE	COD_NODE	SEQUENCIA	TIPO_VALOR	CAMPO_QRY
1072	exercicioLicitacao	173	1	1	integer	ANO_PROC
1073	numeroModalidade	173	1	2	integer	NUM_LIC
1074	dataEditalOuConvite	173	1	3	string	DT_EMISSAO
1075	dataRecebimentoDoc	173	1	4	string	DT_RECEBIMENTO
1076	objeto	173	1	5	string	OBJETO_LICITACAO
1077	numeroDeConvidados	173	1	6	integer	NUM_CONVIDADOS
1078	clausulaProrrogacaoContrato	173	1	7	string	CLAUSULA_PRORROGA
1079	prazoExecucao	173	1	8	integer	PRAZO_ENTREGA
1080	formaPagamento	173	1	9	string	FORMA_PAGAMENTO
1081	criterioDesempateMEEPP	173	1	10	boolean	DESEMPATE_MEEPP
1082	destinacaoExclusivaMEEPP	173	1	11	boolean	EXCLUSIVA_MEEPP
1083	subcontratacaoMEEPP	173	1	12	boolean	SUBCONTRATO_MEEPP
1084	limitePercObjetoContratacaoMEEPP	173	1	13	boolean	PERCENT_OBJ_MEEPP
1085	codModalidadeLicitacao	173	1	14	integer	COD_MODALIDADE
1086	codNaturezaProcedimento	173	1	15	integer	NAT_PROCEDIMENTO
1087	codTipoLicitacao	173	1	16	integer	TIPO_LICITACAO
1088	codRegimeExecucaoObras	173	1	17	integer	REG_EXEC_OBRA
1089	codUnidadeMedidaPrazoExecucao	173	1	18	integer	UD_PRAZO_ENTREGA
1090	processoPorLote	173	1	19	boolean	IND_LOTE
1091	codUnidadeOrcamentaria	173	1	20	integer	COD_UNID_ORC
1092	numeroProcesso	173	1	21	string	NUM_PROC
1093	codNaturezaObjeto	173	1	22	integer	COD_NAT_OBJETO
1094	dataAbertura	173	1	23	string	DT_ABERTURA
1095	idDocumentoPDF	173	1	24	string	ID_EDITAL_ENVIADO
1096	codTipoEnvio	173	1	25	integer	COD_TIPO_ENVIO
1097	motivoAtualizacaoCorrecao	173	1	26	string	MOTIVO_ATUALIZACAO
1098	codOrgaoResponsavel	173	1	27	integer	COD_ORGAO_RESP

Fonte: Autor.

3.5 GERAÇÃO DOS DOCUMENTOS *JSON*

Após as etapas de carga e mapeamento do esquema, inserção das consultas *SQL* associadas aos nós, e vinculação entre os campos destas consultas aos atributos correspondentes de cada nó, a geração dos documentos JSON ocorre de forma automatizada. Para isso, o usuário informará o parâmetro da consulta *SQL* do nó raiz, primeiro nó na hierarquia do esquema, e "disparará" a geração. A partir deste ponto todas as consultas são executadas sequencialmente, seguindo a árvore hierárquica dos nós, e na sequência exata em que as propriedades estruturais são descritas no esquema mapeado. A Figura 17 ilustra a execução do algoritmo responsável pela geração dos documentos JSON, instâncias dos esquemas mapeados.

Figura 17 - Sequência recursiva de execução das consultas



Fonte: Autor.

Através do diagrama é possível observar a sequência de execução das chamadas recursivas. A geração dos documentos JSON instanciados a partir do esquema, terá início pelo nó raiz, passando o parâmetro *id_proc* para seus descendentes diretos. Sequencialmente e de forma automática todas as consultas de todos os nós são disparadas. O algoritmo recursivo, na medida que avança, forma os conjuntos chave-valor previstos para cada nó e os anota, construindo o documento JSON resultante. O algoritmo recursivo desenvolvido para a geração dos documentos se encontra na seção 4.3 do próximo capítulo.

3.6 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Neste capítulo foram abordados os procedimentos propostos. A compreensão destes procedimentos, as sequências de execução dos mesmos, bem como os algoritmos descritos no capítulo seguinte, constituem a principal contribuição deste trabalho. Iniciando pelo processo de carga de esquemas JSON, foi possível mapeá-los para tabelas relacionais. Por meio de processos executados de forma automática, foi preservada a integridade da hierarquia estrutural dos esquemas carregados. Deste modo, a geração de documentos JSON poderá deixar de ser preocupação dos usuários, já que na solução proposta, tais documentos são gerados de modo

automatizado. Uma vez que o escopo dos dados esteja definido no esquema JSON, os usuários poderão concentrar esforços na implementação das consultas *SQL*. Estas deverão ser elaboradas para atender a demanda de informações previstas no esquema, constituindo tarefa de extrema importância, pois são responsáveis por prover os valores que serão anotados nos documentos gerados.

Neste trabalho, a carga dos esquemas JSON, bem como a geração dos documentos JSON foram automatizadas por meio de algoritmos recursivos complexos. Independentemente da complexidade e quantidade de níveis dos esquemas, os algoritmos implementados são capazes de realizar o processamento e gerar os resultados esperados. Estes algoritmos responsáveis por mapear os esquemas e gerar os documentos semiestruturados se encontram detalhados no capítulo seguinte.

4 ALGORITMOS PROPOSTOS

Complementando o capítulo anterior, este capítulo descreve os algoritmos desenvolvidos para o processamento das etapas que constituem a proposta deste trabalho. Os algoritmos desenvolvidos são: Algoritmos para a resolução das referências (Algoritmo 1), Algoritmos responsáveis pelo mapeamento dos esquemas (Algoritmo 2), Algoritmo responsável por preservar a hierarquia dos nós (Algoritmo 3), Algoritmo que define o relacionamento entre nós e atributos (Algoritmo 4) e Algoritmo responsável por gerar documentos JSON (Algoritmo 5).

4.1 ALGORITMO PARA RESOLUÇÃO DE REFERÊNCIAS

Antes de realizar o mapeamento de um esquema, são localizadas as propriedades que contenham definições referenciadas. A resolução das referências consiste em copiar estas definições anotando-as nas propriedades que as referenciam. Este procedimento facilita o processamento da análise das propriedades do esquema, pois sem as referências não há necessidade de localizar definições durante o processamento. O Algoritmo a seguir é responsável por processar a resolução das referências (Figura 18).

Figura 18 – Algoritmo para a resolução das referências

Algoritmo 1: Resolução das Referências entre as Propriedades e suas definições
Entrada: jsonSchema {Esquema Json que poderá possuir referenciamentos}
jsonObj{Objeto Json instancido pelo *parser* inicial do esquema (LkJSON)}
Saída: jsonSchemaRes {Esquema Json resultante após solucionadas as referências}

```
1. numRef ← CountRefOccur(jsonSchema)
2. while numRef > 0 do
3. begin
4.   refPath ← ExtractRefPath(jsonSchema)
5.   RefContent ← CopyRefContent(jsonObj, refPath)
6.   jsonSchemaRes ← Replace(jsonSchemaRes,'"$ref":"'+ refPath +'"',
                            RefContent)
7. end
8. Retorna jsonSchemaRes
```

Fonte: Autor.

O Algoritmo proposto recebe como entrada um esquema JSON e percorre todas as suas propriedades em busca de referenciamentos. Inicialmente o algoritmo calcula a quantidade de

referências existentes no esquema (linha 1). Para cada referência encontrada (linha 2), é obtido o caminho da definição correspondente (linha 4). A definição é copiada (linha 5) e o referenciamento é então substituído pelo texto correspondente (linha 6).

Ao encontrar a palavra-chave *$ref* associada a palavra-chave *items*, obtém o caminho (valor de *$ref)*, encontrando a localização exata da definição referenciada. Após localizar a definição referenciada, substitui a referência pelo trecho de anotação correspondente. O processamento continua até que todas as referências sejam resolvidas. Um exemplo de propriedades referenciadas pode ser observado na Seção 3.2.2.

Ao término desta etapa, a saída obtida é o mesmo esquema JSON, porém em uma versão cujas propriedades possuem itens definidos na própria anotação da propriedade, portanto sem a utilização de referenciamento.

Os Algoritmos a seguir (Figura 19) detalham os métodos denominados *CountRefOccur*, *ExtractRefPath*, e *CopyRefContent*.

Figura 19 – Algoritmo para a contagem das ocorrências de referenciamento

Algoritmo 1.1: CountRefOccur **-** Contagem das Referências do Esquema
Entrada: jsonSchema {Esquema Json que poderá possuir referenciamentos}
Saída: numRef {quantidade de referências encontradas}

```
1. textLen ← Length(jsonSchema)
2. searchedTextLen ← Length('"$ref"')
3. if (textLen <> 0) and (Pos('"$ref"', jsonSchema) <> 0) then
4.     numRef ← (textLen - Length(Replace(jsonSchema,'"$ref"','',
5.                         [rfReplaceAll]))) div searchedTextLen
6. Retorna numRef
```

Fonte: Autor.

Este algoritmo obtém a quantidade de referências encontradas no esquema, sem que seja necessário percorrer explicitamente todo o esquema. A diferença entre o tamanho original do esquema e o tamanho deste mesmo esquema após exclusão de todas as ocorrências da palavra chave "$ref", quando dividido pelo tamanho da string "$ref", resulta no número de referências encontradas neste esquema.

A Figura 20 ilustra o algoritmo de extração do caminho para a definição referenciada.

Figura 20 – Algoritmo para a extração do caminho para a definição

Algoritmo 1.2: ExtractRefPath – Extração do caminho para a definição referenciada

Entrada: jsonSchema {Esquema Json que poderá possuir referenciamentos}
Saída: refPath {caminho para a definição referenciada}

```
1. InitMarkerLength ← Length('"$ref":"')
2. initPosition ← PosEx('"$ref":"', jsonSchema, 1) + InitMarkerLength
3. endPosition ← PosEx('"', jsonSchema, initPosition + InitMarkerLength)
4. refPath ← Copy(jsonSchema, initPosition, endPosition - initPosition)
5. Retorna refPath
```

Fonte: Autor.

O caminho é obtido através da cópia do trecho entre a posição da ocorrência da *string* "$ref": (linha 2) e o primeiro espaço em branco subsequente (linha 3).

Este algoritmo é executado sempre que uma referência é encontrada, repetindo a execução sequencialmente após a resolução da referência anterior, desta forma sempre irá extrair o caminho da referência que esteja sendo processada no momento.

A Figura 21 ilustra o Algoritmo que obtem as definições referenciadas.

Figura 21 – Algoritmo para a obtenção da definição referenciada

Algoritmo 1.3: CopyRefContent – Obtenção da definição referenciada

Entrada: jsonObj {Objeto Json instancido pelo *parser* inicial do esquema (LkJSON)}
refPath {caminho para a definição referenciada}
Saída: jsonSchemaRef {Trecho referenciado obtido através do caminho refPath}

```
1. nodeList ← GetNodeListByPath(refPath)
2. tmpObject ← jsonObj.Field[nodeList[1]]
3. for i ← 2 to nodeList.Count - 1 do
4. begin
5.    tmpObject ← tmpObject.Field[nodeList[i]]
6. end
7. objectName ← nodeList[i-1]
8. jsonSchemaRef ← '"'+ objectName + '":' + GenerateReadableText(tmpObject,i)
9. Retorna jsonSchemaRef
```

Fonte: Autor.

Para a obtenção da definição referenciada, o algoritmo 1.3 recebe o *JSONObj* obtido através do *parser* inicial do esquema, e o caminho da referência a ser encontrada. O caminho *refPath* é então particionado em uma lista de nós (linha 1). A lista de nós auxilia a percorer

sequencialmente as propriedades do objeto json até que a definição seja encontrada (linhas 3 a 6). A definição, após encontrada, é anotada como valor para um novo atributo, este atributo possui nome idêntico ao da propriedade correspondente. A função denominada *GenerateReadableText* formata (indenta) a definição da propriedade.

A obtenção da lista de nós de um caminho (Figura 22) está detalhada no algoritmo a seguir.

Figura 22 – Algoritmo para a obtenção da lista de nodes caminhos da referência

Algoritmo 1.3.1: GetNodeListByPath – Obtenção da lista de nós (nodes) de um caminho de referência

Entrada: refPath {caminho para a definição referenciada}
Saída: nodeList {Lista sequencial de nós de um caminho}

```
1.  lengthRefPath ← Length(refPath)
2.  for i ← 1 to lengthRefPath do
3.  begin
4.     if (Copy(refPath, i, 1) = '/') or
5.                  (i = lengthRefPath) then
6.     begin
7.        if (i = lengthRefPath) then
8.           strItem ← strItem + text[i]
9.        nodeList.Add(strItem)
10.       strItem ← ''
11.    end
12.    else
13.       strItem ← strItem + text[i]
14. end
15. Retorna nodeList
```

Fonte: Autor.

A lista de nós é obtida percorrendo os caracteres do caminho *refPath* em busca de caracteres de barra (linhas 2 a 14). Enquanto *refPath* é percorrido, todos os seus caracteres são concatenados na variável *strItem* até que uma barra (*/*) ou o final de *refPath* seja encontrado (linha 4), sendo então *strItem* inserido em *nodeList* (linha 9). Desta forma, sequencialmente, todos os nós que compõem *refPath* são adicionados à lista.

4.2 ALGORITMOS RESPONSÁVEIS PELO MAPEAMENTO DOS ESQUEMAS JSON

O algoritmo a seguir insere na tabela NODES o primeiro nó fixo, raiz (*root*) do esquema. Logo após, este mesmo algoritmo invoca o algoritmo recursivo denominado *GetNodesDetails*. Este por sua vez percorrerá os demais nós e propriedades armazenando-os nas tabelas correspondente.

Figura 23 – Algoritmo responsáveis pelo início do mapeamento dos Esquema JSON

Algoritmo 2: ImportSchemaStruct – Importa o *Esquema JSON* para o modelo relacional proposto

Entrada: JSONSchemaRes {Esquema *JSON* resultante do solucionamento das referências}
Saída: Tabelas NODES e ATRIBUTOS do modelo relacional, contendo os nós (propriedades estruturais) e as propriedades simples do esquema

```
1. JSONObj ← JSON.ParseText(JSONSchemaRes) as JSONobject
2. idSchema ← GetColValue('SCHEMA', 'ID_SCHEMA_ARQUIVO')
3. cdsSchemaNodes.Append
4. cdsSchemaNodes.Field ('COD_NODE').AsInteger ← 1
5. cdsSchemaNodes.Field ('ROTULO_NODE').AsString ← 'root'
6. cdsSchemaNodes.Field('TIPO_NODE').AsString ← 'object'
7. cdsSchemaNodes.Field('DESCR_NODE').AsString ← 'Nó raiz do arquivo'
8. cdsSchemaNodes.Field('COD_NODE_SUP').Value ← null
9. cdsSchemaNodes.Field('SEQUENCIA').AsInteger ← 1
10. cdsSchemaNodes.Field('ID_SCHEMA_ARQUIVO').AsInteger ← PosEstrutura
11. cdsSchemaNodes.Insert
12. GetNodesDetails(JSONObj,x,propNum,tree);
13. Resulta na inserção de registros no banco relacional (tabelas NODES e ATRIBUTOS)
```

Fonte: Autor.

O algoritmo 2, representado na Figura 23, utiliza uma função de sintática de texto (*ParseText)* que recebe um documento JSON, neste caso o esquema após a resolução das referências, e o converte para um objeto JSON. Desta forma é possível navegar entre suas propriedades em busca de palavras chaves que determinam as características de cada propriedade. O *cdsSchemaNodes* representa um objeto que armazena os dados em *cache*, somente persistindo os mesmos em banco de dados quando sua função *Insert* é executada.

O algoritmo recursivo *GetNodesDetails* (Figura 24), detalhado a seguir, realiza o mapeamento e armazenamento, em banco relacional, das propriedades estruturais (nós) e propriedades simples (atributos associados aos nós) encontrados no esquema JSON.

Figura 24 – Algoritmo responsáveis pelo mapeamento dos *Esquema* JSON

Algoritmo 2.1: GetNodesDetails – Mapeamento e armazenamento das propriedades estruturais (nós) e propriedades simples (atributos associados aos nós)

Entrada: *JSONObj* {Objeto *JSON* resultante da conversão do esquema *JSON* em objeto por meio da função "ParseText"}
propNum {Variável que armazena o número da propriedade}
level {Variável que armazena o nível na estrutura em árvore do esquema}
tree {Variável que armazena os nós percorridos}

Saída: Tabelas NODES e ATRIBUTOS do modelo relacional, contendo os nós (propriedades estruturais) e as propriedades simples do esquema

```
1.  NodePropertyTmp ← TNodeProperty.Create
2.  for i ← 0 to JSONObj.GetField('properties').Count -1 do
3.     isNode ← False
4.     JSONObjTmp ← JSONObj.GetField('properties').Child[i]
5.     for y ← 0 to JSONObj.Field['properties'].field[JSONObjTmp.Name]. Count -1 do
6.        aStr ←  JSONObj.Field['properties'].field[JSONObjTmp.Name].Child[y]).Name
7.        if (aStr ← 'type') then
8.           if ((varToStr(JSONObj.Field['properties'].field[
                      JSONObjTmp.Name].Field[aStr].Value) = 'array') or
                 (varToStr(JSONObj.Field['properties'].field[
                      JSONObjTmp.Name].Field[aStr].Value) = 'object')) then
9.              isNode ← True
10.             Inc(level,1)
11.             tree.Add(IntToStr(level))
12.             Node ← TNode.Create;
13.             Node.nodeLabel ← JSONObjTmp.Name
14.             if varToStr(JSONObj.Field['properties'].field[
                        JSONObjTmp.Name].Field[aStr].Value) = 'array' then
15.                Node.nodeType ← 'array'
16.             else
17.                Node.nodeType ← 'object'
18.             end if
19.          else
20.             if not Assigned(NodeProperty) then
21.                NodeProperty ← TNodeProperty.Create
22.             end if
23.             NodeProperty.propertyLabel ← JSONObjTmp.Name
24.             NodeProperty.propertyType ← varToStr(JSONObj.Field['properties'].
25.                   field[JSONObjTmp.Name].Field['type'].Value)
26.          end if
27.       end if
28.       if (aStr = 'format') then
29.          NodeProperty.propertyFormat ← varToStr(JSONObj.Field['properties'].
                      field[JSONObjTmp.Name].Field['format'].Value)
30.       else
31.          NodeProperty.propertyFormat ← ''
32.       end if
33.       if isNode then
34.          if aStr = 'items' then
35.             JSONObjTmp2 ← JSONObj.field['properties'].field[JSONObjTmp.Name].
```

(Continuação)

```
                    field[aStr].Field[JSONObjTmp.Name]
36.         else
37.         if aStr = Node.nodeLabel then
38.             JSONObjTmp2 ← JSONObj.field['properties'].field[JSONObjTmp.Name].
                    field[aStr]
39.         end if
40.       end if
41.   end for
42.   if isNode then
43.       Node.nodeId ← StrToInt(tree[tree.Count-1])
44.       PosNode ← Node.nodeId
45.       if tree.Count-1 > 0 then
46.           Node.nodeUpperId ← StrToInt(tree[tree.Count-2])
47.       end if
48.       GetNodesDetails(JSONObjTmp2, level, propNum, tree)
49.       Node.nodePropertyList.Add(NodeProperty)
50.       tree.Delete(tree.Count - 1)
51.       idSchema ← cdsSchemaArquivo.Field('ID_SCHEMA_ARQUIVO').AsInteger
52.       cdsSchemaNodes.Append
53.       cdsSchemaNodes.Field('COD_NODE').AsInteger ← Node.nodeId
54.       cdsSchemaNodes.Field('ROTULO_NODE').AsString ← Node.nodeLabel
55.       cdsSchemaNodes.Field('COD_NODE_SUP').AsInteger ← Node.nodeUpperId
56.       cdsSchemaNodes.Field('TIPO_NODE').AsString ← Node.nodeType
57.       cdsSchemaNodes.Field('SEQUENCIA').AsInteger ← Node.nodeId
58.       cdsSchemaNodes.Field('ID_SCHEMA_ARQUIVO').AsInteger ← idSchema
59.       cdsSchemaNodes.Insert
60.   else
61.       Inc(propNum,1)
62.       NodeProperty.propertyId ← propNum
63.       NodeProperty.propertyPosition ← propNum
64.       NodeProperty.nodeId ← StrToInt(tree[tree.Count-1])
65.       if not Assigned(Node) then
66.           Node ← TNode.Create
67.       end if
68.       if Assigned(Node) then
69.         Node.nodePropertyList.Add(NodeProperty)
70.       end if
71.   end if
72.   NodePropertyTmp ← Node.nodePropertyList.Items[Node.nodePropertyList.Count - 1]
73.   if not(Node.nodeLabel <> '') then
74.     if PosNode = 0 then
75.           PosNode ← 1
76.     end if
77.     cdsNodeAtributos.Append
78.     cdsNodeAtributos.Field('ROTULO_ATRIBUTO').AsString ←
                                        NodePropertyTmp.propertyLabel
79.     cdsNodeAtributos.Field('COD_NODE').AsInteger ← PosNode
80.     cdsNodeAtributos.Field('ID_SCHEMA_ARQUIVO').AsInteger ← PosEstrutura
81.     cdsNodeAtributos.Field('SEQUENCIA').AsInteger ←
                                        NodePropertyTmp.propertyPosition
82.     cdsNodeAtributos.Field('TIPO_VALOR').AsString ←
```

(Continuação)

```
                                                    NodePropertyTmp.propertyType
83.       cdsNodeAtributos.Field('FORMATO_VALOR').AsString ←
                                                    NodePropertyTmp.propertyFormat
84.       cdsNodeAtributos.Insert
85.     end if
86. end for
87. NodeProperty.Free
88. Node.Free
```

Fonte: Autor.

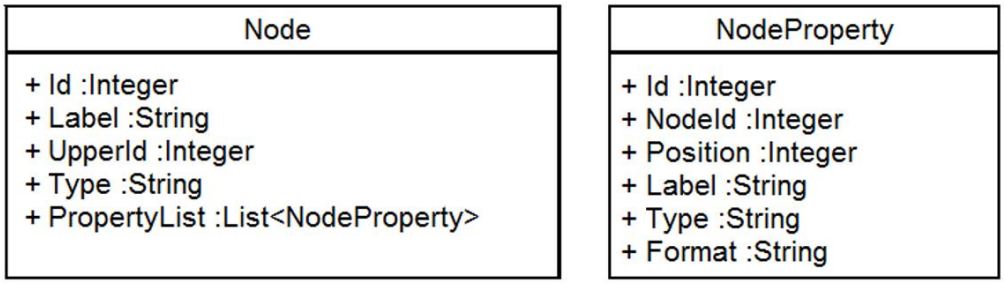
O Algoritmo detalhado acima percorre todas as propriedades do esquema (linha 2), sendo que para cada uma destas propriedades, percorre todos os seus elementos (linha 5) em busca do valor para o elemento *type* (linhas 7). Baseando-se no tipo de propriedade, segrega-as em nós e atributos. Nós são propriedades estruturais do tipo *array* ou *object* (linha 8), para estas propriedades são instanciados objetos da classe *TNode* (linha 12), sendo esta criada para armazenar em memória as características dos nós, são estas: *Id, UpperId*, *type* e *label*. A classe *TNode* armazena ainda uma lista de objetos da classe *TNodeProperty*. Os atributos dos nós são propriedades simples do tipo *number, string, boolean*, entre outras. O algoritmo em questão considera que, todas as propriedades que não são do tipo *array* ou *object*, são atributos associados a um determinado nó (linha 19). Para estes atributos são instanciados objetos da classe *TNodeProperty* (linha 21), esta classe fora criada para comportar as características destes atributos, tais como: *label*, *position*, *type* e *format*.

O processamento das propriedades ocorre na ordem em que se encontram no esquema, sempre que um nó é encontrado, o algoritmo em questão invoca a si mesmo de forma recursiva (linha 48). A recursividade possibilita que todos os nós e propriedades sejam mapeados, independentemente da quantidade de níveis hierárquicos existentes. A medida que as propriedades são percorridas, as mesmas são mapeadas e persistidas no banco de dados relacional, populando as tabelas NODES e ATRIBUTOS.

No quadro 11 encontram-se detalhadas as classes *TNode* e *TNodeProperty*. Tais classes são utilizadas para o armazenamento temporário das características dos nós e atributos do esquema, antes de serem persistidos no banco relacional.

Quadro 11 - Classes utilizadas para armazenamento durante o pré-processamento

Classes utilizadas para o armazenamento temporário das Propriedades Estruturais (Nodes) e Propriedades Simples (Node Properties)



Fonte: Autor.

A execução do Algoritmo 2.1, *GetNodesDetails*, resulta na inserção dos registros correspondentes aos nós e atributos originados do esquema JSON. Os Quadros 12 e 13 demonstram a disposição dos dados em NODES e ATRIBUTOS (alguns campos foram ocultados).

Quadro 12 – Dados dos nós persistidos em banco relacional

SCHEMA_NODES

ID_NODE	COD_NODE	ROTULO_NODE	ID_NODE_SUP	COD_NODE_SUP	TIPO_NODE	SEQUENCIA
173	1	root	(null)	(null)	object	1
175	2	lote	(null)	1	array	2
174	3	item	(null)	2	array	3
176	4	responsaveisPelaLicitacao	(null)	1	array	4
177	5	parecer	(null)	1	array	5
179	6	comissao	(null)	1	array	6
178	7	membro	(null)	6	array	7
181	8	recursoOrcamentario	(null)	1	array	8
180	9	dotacao	(null)	8	array	9
182	10	publicacao	(null)	1	array	10

Fonte: Autor.

Quadro 13 – Dados dos atributos persistidos em banco relacional

NODE_ATRIBUTOS

ID_ATRIBUTO	ROTULO_ATRIBUTO	ID_NODE	COD_NODE	SEQUENCIA	TIPO_VALOR	FORMATO_VALOR	CAMPO_QRY
1072	exercicioLicitacao	(null)	1	1	integer	(null)	(null)
1073	numeroModalidade	(null)	1	2	integer	(null)	(null)
1074	dataEditalOuConvite	(null)	1	3	string	date	(null)
1075	dataRecebimentoDoc	(null)	1	4	string	date	(null)
1076	objeto	(null)	1	5	string	(null)	(null)
1077	numeroDeConvidados	(null)	1	6	integer	(null)	(null)
1078	clausulaProrrogacaoContrato	(null)	1	7	string	(null)	(null)
1079	prazoExecucao	(null)	1	8	integer	(null)	(null)
1080	formaPagamento	(null)	1	9	string	(null)	(null)
1081	criterioDesempateMEEPP	(null)	1	10	boolean	(null)	(null)
1082	destinacaoExclusivaMEEPP	(null)	1	11	boolean	(null)	(null)
1083	subcontratacaoMEEPP	(null)	1	12	boolean	(null)	(null)
1084	limitePercObjetoContratacaoMEEPP	(null)	1	13	boolean	(null)	(null)
1085	codModalidadeLicitacao	(null)	1	14	integer	(null)	(null)
1086	codNaturezaProcedimento	(null)	1	15	integer	(null)	(null)
1087	codTipoLicitacao	(null)	1	16	integer	(null)	(null)
1088	codRegimeExecucaoObras	(null)	1	17	integer	(null)	(null)
1089	codUnidadeMedidaPrazoExecucao	(null)	1	18	integer	(null)	(null)
1090	processoPorLote	(null)	1	19	boolean	(null)	(null)
1091	codUnidadeOrcamentaria	(null)	1	20	integer	(null)	(null)
1092	numeroProcesso	(null)	1	21	string	(null)	(null)
1093	codNaturezaObjeto	(null)	1	22	integer	(null)	(null)
1094	dataAbertura	(null)	1	23	string	date	(null)
1095	idDocumentoPDF	(null)	1	24	string	arquivo-ged	(null)
1096	codTipoEnvio	(null)	1	25	integer	(null)	(null)
1097	motivoAtualizacaoCorrecao	(null)	1	26	string	(null)	(null)
1098	codOrgaoResponsavel	(null)	1	27	integer	(null)	(null)
1099	numeroLote	(null)	2	28	integer	(null)	(null)
1100	descricaoLote	(null)	2	29	string	(null)	(null)
1101	numeroItem	(null)	3	30	integer	(null)	(null)
1102	quantidade	(null)	3	31	number	(null)	(null)

Fonte: Autor.

Durante o processamento do Algoritmo 2.1, as colunas COD_NODE e COD_NODE_SUP de NODES foram “alimentadas” mantendo a estrutura hierárquica existente entre os nós. Porém para cada esquema *JSON* carregado, a coluna COD_NODE terá o valore reiniciado, desta forma haverá repetição entre os valores desta coluna para esquemas distintos. Para garantir a integridade referencial dos nós armazenados, o Algoritmo 3 (Figura 25), apresentado a seguir, é responsável pela execução de um *script SQL* que definirá a coluna ID_NODE_SUP, campo chave estrangeira que referencia ID_NODE.

Figura 25 – Algoritmo que preserva a hierarquia dos nodes

Algoritmo 3: SetHierarchyOfNodes – Define o relacionamento entre os campos ID_NODE e ID_NODE_SUP preservando assim a hierarquia dos nós

Entrada: idSchemaArquivo {valor do campo ID_SCHEMA_ARQUIVO, identificador exclusivo do esquema na tabela SCHEMA}
Saída: Relacionamento entre os campos ID_NODE e ID_NODE_SUP definidos na tabela NODES

(Continuação)

```
1.  MetaQry.SQL.Text ←
2.     'UPDATE NODES SN ' +
3.     'SET SN.ID_NODE_SUP = (SELECT SN2.ID_NODE ' +
4.     '                      FROM NODES SN2 ' +
5.     '                      WHERE SN2.COD_NODE = SN.COD_NODE_SUP ' +
6.     '                            AND SN2.ID_SCHEMA_ARQUIVO =  ' +
7.     '                                SN.ID_SCHEMA_ARQUIVO ' +
8.     '                     )  ' +
9.     'WHERE SN.ID_SCHEMA_ARQUIVO = :idSchemaArquivo '
10. MetaQr.ParamByName('idSchemaArquivo').AsInteger ← idSchemaArquivo
11. ExecuteQry(MetaQry)
12. Resulta na preservação da hierarquia dos nós extraídos do esquema
```

Fonte: Autor.

O Quadro 14 demonstra os registros resultantes do processamento do Algoritmo 3, após a definição do auto relacionamento existente na tabela NODES por meio dos campos ID_NODE e ID_NODE_SUP.

Quadro 14 – Hierarquia dos nós preservada no banco relacional

SCHEMA_NODES

ID_NODE	COD_NODE	ROTULO_NODE	**ID_NODE_SUP**	COD_NODE_SUP	TIPO_NODE	SEQUENCIA
173	1	root	**(null)**	(null)	object	1
175	2	lote	**173**	1	array	2
174	3	item	**175**	2	array	3
176	4	responsaveisPelaLicitacao	**173**	1	array	4
177	5	parecer	**173**	1	array	5
179	6	comissao	**173**	1	array	6
178	7	membro	**179**	6	array	7
181	8	recursoOrcamentario	**173**	1	array	8
180	9	dotacao	**181**	8	array	9
182	10	publicacao	**173**	1	array	10

Fonte: Autor.

O algoritmo 2.1, resolve os nós a partir de suas extremidades, desta forma atributos são inseridos na tabela ATRIBUTOS, antes dos nós em NODES. Desta forma não é alimentada a coluna ID_NODE, chave estrangeira entre ATRIBUTOS e NODES. Para garantir a integridade referencial o algoritmo 4, detalhado na Figura 26, realiza o processo de *update* da coluna ID_NODE com base na coluna COD_NODE.

Figura 26 – Algoritmo que processa o relacionamento entre os nós e atributos

Algoritmo 4: SetNodeIds – Define o relacionamento entre as tabelas NODES e ATRIBUTOS através do campo ID_NODE

Entrada: idSchemaArquivo {valor do campo ID_SCHEMA_ARQUIVO, identificador exclusivo do esquema na tabela SCHEMA}
Saída: Relacionamento entre entre as tabelas NODES e ATRIBUTOS através do campo ID_NODE

```
1.  MetaQry.SQL.Text ←
2.     'UPDATE ATRIBUTOS NA ' +
3.     'SET NA.ID_NODE = (SELECT SN.ID_NODE ' +
4.     '                        FROM NODES SN ' +
5.     '                        WHERE SN.COD_NODE = NA.COD_NODE AND ' +
6.     '                                SN.ID_SCHEMA_ARQUIVO = ' +
7.     '                                NA.ID_SCHEMA_ARQUIVO ' +
8.     '                      ) ' +
9.      'WHERE NA.ID_SCHEMA_ARQUIVO = :idSchemaArquivo '
10. MetaQr.ParamByName('idSchemaArquivo').AsInteger ← idSchemaArquivo
11. ExecuteQry(MetaQry)
12. Resulta na correta definição da chave estrangeira ID_NODE da tabela ATRIBUTOS
```

Fonte: Autor.

O Quadro 15 demonstra os registros resultantes do processamento do Algoritmo 4, após a definição da chave estrangeira ID_NODE.

Quadro 15 – Integridade referencial preservada no banco relacional

NODE_ATRIBUTOS

ID_ATRIBUTO	ROTULO_ATRIBUTO	ID_NODE	COD_NODE	SEQUENCIA	TIPO_VALOR	FORMATO_VALOR	CAMPO_QRY
1072	exercicioLicitacao	173	1	1	integer	(null)	(null)
1073	numeroModalidade	173	1	2	integer	(null)	(null)
1074	dataEditalOuConvite	173	1	3	string	date	(null)
1075	dataRecebimentoDoc	173	1	4	string	date	(null)
1076	objeto	173	1	5	string	(null)	(null)
1077	numeroDeConvidados	173	1	6	integer	(null)	(null)
1078	clausulaProrrogacaoContrato	173	1	7	string	(null)	(null)
1079	prazoExecucao	173	1	8	integer	(null)	(null)
1080	formaPagamento	173	1	9	string	(null)	(null)
1081	criterioDesempateMEEPP	173	1	10	boolean	(null)	(null)
1082	destinacaoExclusivaMEEPP	173	1	11	boolean	(null)	(null)
1083	subcontratacaoMEEPP	173	1	12	boolean	(null)	(null)
1084	limitePercObjetoContratacaoMEEPP	173	1	13	boolean	(null)	(null)
1085	codModalidadeLicitacao	173	1	14	integer	(null)	(null)
1086	codNaturezaProcedimento	173	1	15	integer	(null)	(null)
1087	codTipoLicitacao	173	1	16	integer	(null)	(null)
1088	codRegimeExecucaoObras	173	1	17	integer	(null)	(null)
1089	codUnidadeMedidaPrazoExecucao	173	1	18	integer	(null)	(null)
1090	processoPorLote	173	1	19	boolean	(null)	(null)
1091	codUnidadeOrcamentaria	173	1	20	integer	(null)	(null)
1092	numeroProcesso	173	1	21	string	(null)	(null)
1093	codNaturezaObjeto	173	1	22	integer	(null)	(null)
1094	dataAbertura	173	1	23	string	date	(null)
1095	idDocumentoPDF	173	1	24	string	arquivo-ged	(null)
1096	codTipoEnvio	173	1	25	integer	(null)	(null)
1097	motivoAtualizacaoCorrecao	173	1	26	string	(null)	(null)
1098	codOrgaoResponsavel	173	1	27	integer	(null)	(null)
1099	numeroLote	175	2	28	integer	(null)	(null)
1100	descricaoLote	175	2	29	string	(null)	(null)
1101	numeroItem	174	3	30	integer	(null)	(null)
1102	quantidade	174	3	31	number	(null)	(null)

Fonte: Autor.

Após a conclusão do processo de carga do esquema JSON, todas as suas propriedades foram analisadas de forma automática e segregadas em nós e atributos destes nós. Os nós e suas características foram inseridos na tabela NODES, preservando a estruturação hierárquica dos mesmos, e a sequência com que são anotados no esquema. Os atributos bem como suas características foram inseridos na tabela ATRIBUTOS, de forma análoga ao que ocorre com os nós, atributos também tiveram a sequência exata mantida. Desta forma foi possível obter o esquema mapeado para o modelo relacional. Neste modelo é facilitada a manipulação de seus elementos, sendo possível a vinculação de consultas *SQL* responsáveis pela recuperação dos dados.

4.3 ALGORITMO GERADOR DE DOCUMENTOS *JSON* (INSTÂNCIAS DE UM ESQUEMA)

Após as etapas de carga e mapeamento do esquema, inserção das consultas *SQL* associadas aos nós, definição de parâmetros, e vinculação dos campos destas consultas aos atributos correspondentes de cada nó, a geração dos documentos JSON ocorre de forma automatizada. Para isso, o usuário informará apenas o parâmetro da consulta do nó raiz. A partir deste ponto todas as consultas são executadas sequencialmente.

O algoritmo a seguir (Figura 27) é responsável pela geração dos documentos JSON, instâncias do esquema mapeado.

Figura 27 – Algoritmo que processa a geração dos documentos JSON

Algoritmo 5: JSONDocGenerate – Algoritmo Recursivo Gerador de Documentos *JSON*

Entradas: idSchemaArquivo {ID do esquema na tabela SCHEMA}
paramValue {valor do parâmetro do nó raiz, primeiro nó do esquema}
JSONDoc {variável - documento JSON em construção}
listTagFechamento {armazenará a lista de caracteres de fechamento dos nós}
listTagFechamentoInt {armazenará a lista de fechamento dos elementos dos vetores}
listParam {armazenará a lista de parâmetros das consultas SQL dos nós}
listParamValue {armazenará a lista de valores dos parâmetros das consultas SQL}
numNode {variável – nº do nó atualmente processado nas "chamadas" recursivas}
aCdsDataSeq {objeto que armazenará em memória os nós do esquema}
cdsData {objeto que receberá uma cópia dos registros dos nós do esquema}

Saída: Documentos *JSON* instâncias do esquema mapeado

```
1.  aBreak ← False
2.  if JSONDoc = '' then
3.    param ← SearchFirstNodeParameter(idSchemaArquivo)
4.    aData ← GetNodesBySchema(idSchemaArquivo)
```

(Continuação)

```
5.      cdsData.Data ← aData
6.      aCdsDataSeq.Data ← aData
7.    end if
8.
9.    if not cdsData.IsEmpty then
10.     while not cdsData.Eof do
11.       if cdsData.RecNo > numNode then
12.         numNode ← cdsData.RecNo
13.       end if
14.       JSONDoc ← JSONDoc + cdsData.Field('INICIO').AsString
15.       listTagFechamento.Add(cdsData.Field('FIM').AsString)
16.       MetaQr.SQL.Text ← cdsData.Field('SCRIPT').AsString
17.
18.       if cdsData.Field('ROTULO_NODE').AsString = 'root' then
19.         if param <> '' then
20.           MetaQr.SQL.Text ← Replace(MetaQr.SQL.Text,param,paramValue)
21.         end if
22.       else
23.         paramEx ← listParam.Strings[listParam.Count - 1]
24.         paramExValue ← listParamValue.Strings[listParamValue.Count - 1]
25.         MetaQr.SQL.Text ← Replace(MetaQr.SQL.Text,':'+paramEx,''''+
                                paramExValue+'''')
26.       end if
27.       aDataTmp ← OpenQuery(MetaQr)
28.       cdsDataTmp.Data ← aDataTmp
29.
30.       MetaQr2.ParamByName('idNode').Value ← cdsData.Field(
                'ID_NODE').AsInteger
31.       aDataTmp2 ← OpenQuery(MetaQr2)
32.       cdsDataTmp2.Data ← aDataTmp2
33.
34.       while not cdsDataTmp.Eof do
35.         JSONDoc ← JSONDoc + cdsData.Field('INICIO_ELEM_
                ARRAY').AsString
36.         listTagFechamentoInt.Add(cdsData.Field('FIM_ELEM_
                ARRAY').AsString)
37.
38.         paramEx ← cdsData.Field('CHAVE_EXT').AsString
39.         idNode ← cdsData.Field('ID_NODE').AsInteger
40.
41.         for i ← 0 to cdsDataTmp.FieldCount - 1 do
42.           JSONDoc ← JSONDoc + cdsDataTmp.FieldList.Fields[i].FieldName +
                        ':' + VarToStr(cdsDataTmp.FieldList.Fields[i].Value)
43.           if i < cdsDataTmp.FieldCount - 1 then
44.             JSONDoc ← JSONDoc + ','
45.           end if
46.         end for
47.
48.         for i ← 0 to cdsDataTmp.FieldCount - 1 do
49.           if cdsDataTmp2.Locate('CAMPO_QRY',cdsDataTmp.
```

(Continuação)

```
                    FieldList.Fields[i].FieldName,[]) then
50.             if cdsDataTmp2.Field('TIPO_VALOR').AsString =
                      'string' then
51.               JSONDoc ← JSONDoc + ''' + cdsDataTmp2.Field(
                          'ROTULO _ATRIBUTO').AsString + ''':''' +
                          VarToStr(cdsDataTmp.FieldList.Fields[i].Value) + '''
52.             else
53.               JSONDoc ← JSONDoc + ''' + cdsDataTmp2.Field(
                          'ROTULO _ATRIBUTO').AsString + ''':' +
                          VarToStr(cdsDataTmp.FieldList.Fields[i].Value)
54.             end if
55.
56.             if i < cdsDataTmp.FieldCount - 1 then
57.               JSONDoc ← JSONDoc + ','
58.             end if
59.           end if
60.         end for
61.
62.         if paramEx <> '' then
63.           paramExValue ← cdsDataTmp.Field(paramEx).AsString
64.         else
65.           paramExValue ← ''
66.         end if
67.         listParam.Add(paramEx)
68.         listParamValue.Add(paramExValue)
69.
70.         if cdsData.RecNo < cdsData.RecordCount then
71.           cdsData.Next
72.           idNodeSup2 ← cdsData.Field('ID_NODE_SUP').AsInteger
73.           cdsData.Prior
74.         end if
75.
76.         if idNodeSup2 = idNode then
77.           JSONDoc ← JSONDoc + ','
78.           if cdsData.RecNo < cdsData.RecordCount then
79.             cdsData.Next
80.             JSONDocGenerate(paramValue, idSchemaArquivo, JSONDoc,
                        listTagFechamento, listTagFechamentoInt, listParam,
                        listParamValue, numNode, aCdsDataSeq, cdsData)
81.           end if
82.         end if
83.
84.         listParam.Delete(listParam.Count - 1)
85.         listParamValue.Delete(listParamValue.Count - 1)
86.         JSONDoc ← JSONDoc + listTagFechamentoInt.Strings[
                        listTagFechamentoInt.Count - 1]
87.         listTagFechamentoInt.Delete(listTagFechamentoInt.Count - 1)
88.
89.         if cdsDataTmp.RecNo < cdsDataTmp.RecordCount then
90.           JSONDoc ← JSONDoc + ','
```

(Continuação)

```
91.             else
92.                 aBreak ← True
93.             end if
94.
95.             cdsDataTmp.Next
96.         end while
97.
98.         JSONDoc ← JSONDoc + listTagFechamento.Strings[listTagFechamento.
                        Count - 1]
99.         listTagFechamento.Delete(listTagFechamento.Count - 1)
100.
101.        if aBreak then
102.        Break
103.        end if
104.
105.        cdsData.Next
106.     end while
107.
108.      aCdsDataSeq.RecNo ← numNode
109.      if cdsData.RecNo < cdsData.RecordCount then
110.          aCdsDataSeq.Next
111.          idNodeSup2 ← aCdsDataSeq.Field('ID_NODE_SUP'
                    ).AsInteger
112.          aCdsDataSeq.Prior
113.      end if
114.
115.      if idNodeSup2 = cdsData.Field('ID_NODE_SUP'
                  ).AsInteger then
116.          if cdsData.RecNo < cdsData.RecordCount then
117.              aCdsDataSeq.Next
118.              JSONDocGenerate(paramValue, idSchemaArquivo, JSONDoc,
                      listTagFechamento, listTagFechamentoInt, listParam,
                      listParamValue, numNode, aCdsDataSeq, aCdsDataSeq)
119.          end if
120.      end if
121.   cdsData.Prior
122.  end if
123.  result ← JSONDoc
124.  Resulta no documento JSON instância do esquema mapeado
```

Fonte: Autor.

O algoritmo recursivo acima é responsável por gerar os documentos JSON. A execução é iniciada pela obtenção do parâmetro da consulta SQL do nó raiz (linha 3), sendo este substituído pelo valor informado pelo usuário (linha 20). Todos os nós, bem como os caracteres de início e fechamento destes, são carregados em memória (linha 4). As consultas SQL associadas aos nós são executadas uma a uma (linha 27), seguindo a sequência exata em que estão dispostos na árvore hierárquica dos mesmos. A árvore hierárquica dos nós é percorrida iniciando pelo nó raiz e avançando para os "galhos" no sentido do centro para as extremidades.

Por meio de “chamadas” recursivas (linhas 80 e 118), cada nó e seus atributos são processados e anotados no documento JSON, este é gerado de forma gradual, no decorrer de cada ciclo de execução recursiva.

O processamento da árvore hierárquica de um documento JSON é de natureza complexa, pois a quantidade de níveis dos relacionamentos entre os nós é imprevisível e dependerá da estruturação do esquema mapeado. Somado a isto, também são imprevisíveis as quantidades de registros que a execução das consultas *SQL* associadas a cada nó irá trazer. A quantidade de registros recuperados por uma consulta de um nó-pai determinará a quantidade de vezes que as consultas dos nós-filhos irão ser executada.

Para a execução da primeira consulta, relacionada ao nó raiz, o usuário informará o valor para o parâmetro. Já as demais consultas serão “disparadas” automaticamente, a parametrização destas se dá por meio do campo chave externa definido no nó-pai. A Figura 28 apresenta o algoritmo responsável pela obtenção do parâmetro da primeira consulta a ser executada.

Figura 28 – Algoritmo que obtém o rótulo do parâmetro do nó raiz

Algoritmo 5.1: SearchFirstNodeParameter – Obtém o parâmetro da Consulta SQL do nó raiz

Entrada: idSchemaArquivo {valor do campo ID_SCHEMA_ARQUIVO, identificador exclusivo do esquema na tabela SCHEMA}
Saída: param {Parâmetro da consulta SQL do nó raiz}

```
1.  MetaQr.SQL.Text ←
2.      ‘SELECT SN.SCRIPT ’ + #13 +
3.      ‘FROM SCHEMA SA’ + #13 +
4.      ‘       INNER JOIN NODES SN’ + #13 +
5.      ‘       ON SA.ID_SCHEMA_ARQUIVO =
                        SA.ID_SCHEMA_ARQUIVO AND’ + #13 +
6.      ‘           SN.SEQUENCIA = 1 AND’ + #13 +
7.      ‘           SA.ID_SCHEMA_ARQUIVO = :idSchema’
8.
9.  MetaQr.SQL.Text ← Replace(MetaQr.SQL.Text,’idSchema’,
                            idSchemaArquivo)
10. aData ← OpenQuery(MetaQr)
11. cdsData.Data ← aData
12.
13. if cdsData.RecordCount > 0 then
14.     param ← Copy (cdsData.Field(‘SCRIPT’).AsString,
                    Pos(‘:’, cdsData.Field(‘SCRIPT’).AsString),
                    length(cdsData.Field(‘SCRIPT’).AsString) –
                    Pos(‘:’, cdsData.Field(‘SCRIPT’).AsString))
15.     param ← Copy (param, 1, Pos(‘ ‘, param)-1)
16.     result ← param
17. end
```

(Continuação)

```
18. else
19.     result ← ‘’
20. end if
21. Resulta no parâmetro da consulta associada ao nó raiz
```

Fonte: Autor.

A obtenção de todos os nós do esquema a ser processado, mapeados previamente para a tabela NODES, é realizada por meio do algoritmo apresentado na Figura 29.

Figura 29 – Algoritmo que obtém os nós de um *squema*

Algoritmo 5.2: GetNodesBySchema – Obtém os nós de determinado esquema
Entrada: idSchemaArquivo {valor do campo ID_SCHEMA_ARQUIVO, identificador exclusivo do esquema na tabela SCHEMA}
Saída: aData {nós e suas características}

```
1.  MetaQr.SQL.Text ←
2.     'WITH COUNT_ATRIB AS (' + #13 +
3.     'SELECT NA.ID_NODE, SN.ROTULO_NODE, COUNT(*) AS
           QTD_ATRIBUTOS' + #13 +
4.     ‘FROM NODES SN' + #13 +
5.     '          INNER JOIN ATRIBUTOS NA' + #13 +
6.     '          ON SN.ID_NODE = NA.ID_NODE AND' + #13 +
7.     '               SN.ID_SCHEMA_ARQUIVO = :idSchema' + #13 +
8.     'GROUP BY NA.ID_NODE, SN.ROTULO_NODE' + #13 +
9.     '), ' + #13 +
10.    'COUNT_NODE_FILHOS AS (' + #13 +
11.    'SELECT SN.ID_NODE, SN.ROTULO_NODE, COUNT(*) AS
             QTD_NODES_FILHOS ' + #13 +
12.    'FROM NODES SN' + #13 +
13.    '          LEFT JOIN NODES SN2' + #13 +
14.    '          ON SN.ID_NODE = SN2.ID_NODE_SUP' + #13 +
15.    'WHERE SN.ID_SCHEMA_ARQUIVO = :idSchema' + #13 +
16.    'GROUP BY SN.ID_NODE, SN.ROTULO_NODE' + #13 +
17.    ')' + #13 +
18.    'SELECT SN.SEQUENCIA, SN.ID_NODE, SN.ID_NODE_SUP, ' + #13 +
19.    '      SN.ROTULO_NODE, CA.QTD_ATRIBUTOS,
             CNF.QTD_NODES_FILHOS,' + #13 +
20.    '      CASE ' + #13 +
21.    '          WHEN SN.TIPO_NODE = "object" AND ' + #13 +
22.    '                   SN.ROTULO_NODE = "root" THEN "{"' + #13 +
23.    '          WHEN SN.TIPO_NODE = "array" AND ' + #13 +
24.    '                   SN.ROTULO_NODE = "root" THEN "["' + #13 +
25.    '          WHEN SN.TIPO_NODE = "object" THEN
                        ‘””’’|| SN.ROTULO_NODE || ‘””:{“’ + #13 +
26.    '          WHEN SN.TIPO_NODE = "array" THEN
                        ‘””’’|| SN.ROTULO_NODE || ‘””:[“’ + #13+
```

(Continuação)

```
27.  '          END AS INICIO,' + #13 +
28.  '     CASE ' + #13 +
29.  '          WHEN SN.TIPO_NODE = "object" AND ' + #13 +
30.  '                    SN.ROTULO_NODE = "root" THEN "}"' + #13 +
31.  '          WHEN SN.TIPO_NODE = "array" AND ' + #13 +
32.  '                    SN.ROTULO_NODE = "root" THEN "}"' + #13 +
33.  '          WHEN SN.TIPO_NODE = "object" THEN "}"' + #13 +
34.  '          WHEN SN.TIPO_NODE = "array" THEN "]"' + #13 +
35.  '     END AS FIM,' + #13 +
36.  '     CASE              ' + #13 +
37.  '        WHEN SN.TIPO_NODE = "array" AND' + #13 +
38.  '        (CA.QTD_ATRIBUTOS + CNF.QTD_NODES_FILHOS) > 1
                                        THEN "{"' + #13 +
39.  '     END AS INICIO_ELEM_ARRAY,' + #13 +
40.  '     CASE              ' + #13 +
41.  '       WHEN SN.TIPO_NODE = "array" AND' + #13 +
42.  '          (CA.QTD_ATRIBUTOS + CNF.QTD_NODES_FILHOS) > 1
                    THEN "}"' + #13 +
43.  '     END AS FIM_ELEM_ARRAY,
44.  '     SN.SCRIPT, SN.CHAVE_EXT    ' + #13 +
45.  'FROM NODES SN' + #13 +
46.  '     INNER JOIN COUNT_ATRIB CA' + #13 +
47.  '     ON SN.ID_NODE = CA.ID_NODE ' + #13 +
48.  '     INNER JOIN COUNT_NODE_FILHOS CNF' + #13 +
49.  '     ON SN.ID_NODE = CNF.ID_NODE' + #13 +
50.  'ORDER BY SN.SEQUENCIA'
51.
52. MetaQr.SQL.Text ← Replace(MetaQr.SQL.Text,'idSchema',
                              idSchemaArquivo)
53. aData ← OpenQuery(MetaQr)
54. Result ← aData
55. Resulta em um DataSet todos os nós e suas características
```

Fonte: Autor.

O algoritmo acima submete uma consulta SQL ao banco relacional, esta retorna todos os nós de um determinado esquema, selecionando-os através da definição do parâmetro *idSchema*. As características de cada nó, bem como os seus caracteres de início e término, são utilizados durante a anotação do documento JSON que estiver sendo gerado. Observa-se que a consulta utiliza visões (*views*) para a obtenção da quantidade de atributos existentes para cada nó (linhas 2 a 9), e a quantidade de nós-filhos (linhas 10 a 17), tais informações são utilizadas para determinar os caracteres de início e término dos elementos dos nós do tipo matriz. O Algoritmo 5.2 obtém ainda as consultas *SQL* que serão executadas durante o processamento da árvore de nós, bem como recupera o campo chave de ligação entre os nós-pai e seus nós-filhos.

5 ESTUDO DE CASO

O notório interesse da população brasileira em conhecer e fiscalizar a utilização do dinheiro público vem impulsionando a popularização dos portais de transparência governamentais. A lei de acesso à informação (LAI), Lei nº 12.527, de 18 de novembro de 2011, tornou obrigatório, aos órgãos e entidades públicas, a divulgação de informações referentes a execução orçamentária, tais como: os procedimentos licitatórios, respectivos editais e resultados. A referida lei determina ainda que os dados sejam disponibilizados em sítios oficiais da Internet, de modo que as informações possam ser acessadas por meios automatizados por sistemas externos e em formatos abertos, estruturados e legíveis por máquina. Atualmente todos os entes governamentais devem manter portais de transparência que evidenciem a arrecadação de receitas e as despesas executadas diariamente (DE PAIVA e REVOREDO, 2016).

O grande volume de informações orçamentárias e financeiras, armazenadas em estruturas complexas, mantidas em bancos de dados relacionais distintos, faz com que a disponibilização das informações de que trata a LAI, configure um desafio aos órgãos e entidades públicas. Tais bancos de dados muitas vezes não possuem suporte nativo aos recursos necessários à exportação de dados nos formatos abertos (W3C – Manual de Dados Abertos) exigidos pelos órgãos fiscalizadores como os tribunais de contas, bem como os portais de transparência do governo. Dentre os formatos de dados abertos existentes, os recomendados para a publicação de dados públicos são os formatos *CSV*, *XML* e *JSON* (Guia de Transparência Ativa Para Órgãos e entidades do Poder Executivo Federal). No intuito de atender integralmente aos preceitos descritos na lei de acesso a informação, bem como possibilitar eficiência no processamento, entidades públicas têm adotando padrões para o envio dos dados a serem auditados e disponibilizados ao cidadão. Embora a LAI não determine qual formato de dado aberto deve ser adotado, é perceptível a crescente utilização da notação JSON como forma de veiculação de tais informações.

Para validar a metodologia proposta foi implementado um protótipo, descrito no Anexo deste trabalho, aplicado em um estudo de caso objetivando a geração de dados abertos. Os dados utilizados originam-se de um banco de dados relacional populado. Tais dados serão dispostos em documentos estruturados na notação JSON, seguindo os padrões exatos definidos pelo Tribunal de Contas dos municípios do estado de Goiás (Portal do Tribunal de Contas dos

Municípios do Estado de Goiás). Porém o processo proposto permite a geração de documentos JSON instanciados a partir de quaisquer esquemas JSON válidos.

Este capítulo demonstra o estudo de caso desenvolvido, estando dividido em três seções. A Seção 5.1 apresenta os esquemas JSON utilizados, a Seção 5.2 descreve o conjunto de dados do modelo relacional utilizado, e a Seção 5.3 apresenta os resultados obtidos.

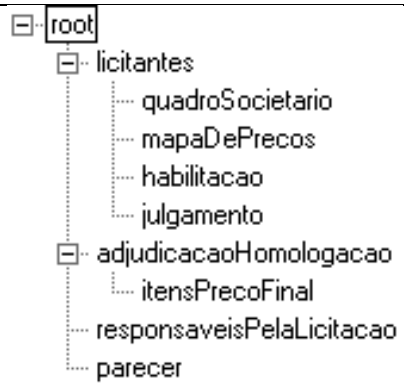
5.1 ESQUEMAS JSON UTILIZADOS

O estudo de caso aplicado utilizou esquemas JSON desenvolvidos pelo TCM. Tais esquemas foram concebidos como padrões para a geração dos documentos para a prestação de contas dos municípios. Estes arquivos deverão conter as informações referentes à execução orçamentária das prefeituras e autarquias dos municípios. Os esquemas JSON completos utilizados neste estudo estão acessíveis, de forma aberta, no portal da prestação de contas eletrônica do TCM (Prestação Eletrônica de Contas do Tribunal de Contas dos Municípios do Estado de Goiás). Foram utilizados dois esquemas referentes a prestação de contas dos processos licitatórios. O Quadro 16 apresentam os principais aspectos estruturais destes esquemas.

Quadro 16 – Esquemas utilizados no estudo de caso e suas propriedades

	Árvore Hierárquica (Estrutura do esquema)	**Tipo de Nó**	**Quant. Atributos**	**Quant. Nós-filho**
LICITACAOFASE1	root	object	27	06
	lote	array	02	01
	item	array	10	--
	responsaveisPelaLicitacao	array	02	--
	parecer	array	03	--
	comissao	array	06	01
	membro	array	03	--
	recursoOrcamentario	array	01	01
	dotacao	array	11	--
	publicacao	array	05	--

(Continuação)

	Árvore Hierárquica (Estrutura do esquema)	Tipo de Nó	Quant. Atributos	Quant. Nós-filho
LICITACAOFASE2		object	03	02
		array	20	04
		array	03	--
		array	04	--
		object	03	--
		object	03	--
		object	04	01
		array	04	--
		array	02	--
		array	03	--

Fonte: Autor.

O esquema denominado LICITACAOFASE1 refere-se à estruturação dos documentos que conterão as informações sobre a fase interna das licitações, compreendendo dados que vão desde a elaboração do edital até sua efetiva publicação. Já no esquema LICITACAOFASE2 são padronizadas as informações relativas aos fornecedores licitantes e os resultados das licitações. As tabelas acima demonstram a árvore de nós dos esquemas utilizados neste estudo.

5.2 CONJUNTO DE DADOS UTILIZADOS

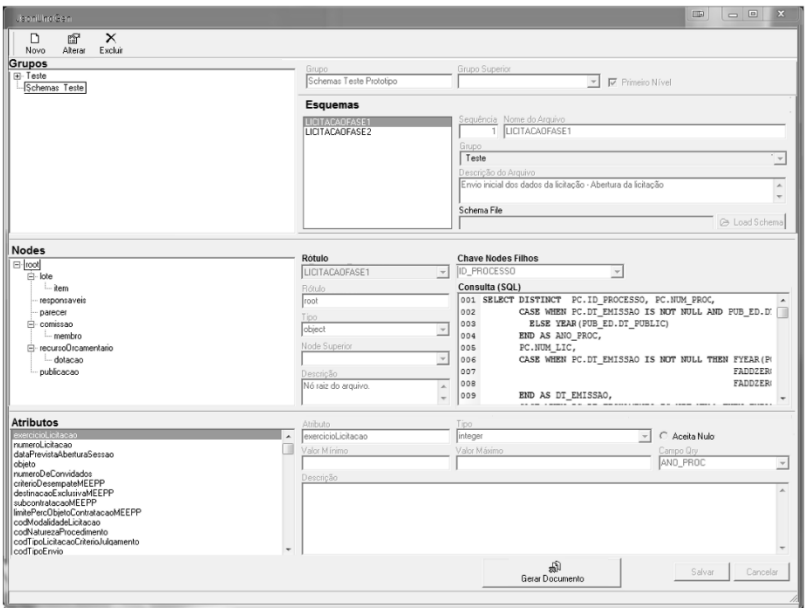
Os dados utilizados neste estudo originam-se de um banco de dados relacional. De um total de 15770 processos de aquisição de bens e serviços, foram selecionados 2080 processos referentes ao ano orçamentário 2018. Para a obtenção das informações previstas nos esquemas do TCM, foram necessárias consultas *SQL* envolvendo junções entre 43 tabelas do modelo relacional. Em conformidade com a metodologia proposta neste trabalho, foram informados como parâmetro para o nó-raiz das árvores hierárquicas dos esquemas, o número do processo de compra (coluna que identifica um processo de compras no banco de dados relacional utilizado).

5.3 RESULTADOS OBTIDOS

Após a obtenção dos esquemas, apresentados anteriormente na Seção 5.1, os mesmos foram carregados por meio da interface do protótipo implementado neste trabalho, sendo esses processados de forma automática. O processamento dos esquemas interpretou cada um dos elementos essenciais do esquema JSON, resultando na classificação dos mesmos em propriedades estruturais e simples. Propriedades estruturais são os objetos e vetores do esquema, estes originam os nós da árvore hierárquica que prevê a disposição dos dados no documento JSON a ser gerado. Elementos simples são as propriedades associadas aos elementos estruturais, tais propriedades podem receber valores do tipo cadeia de caracteres, numerais, boleanos ou nulo. Estes elementos definem as chaves que receberão os valores dos dados contidos no documento resultante.

A cada propriedade estrutural, consideradas nós da árvore hierárquica do esquema, foi associada uma consulta *SQL* responsável pela recuperação dos dados originados das tabelas estruturais. Os campos da projeção destas consultas foram vinculados aos elementos simples do esquema carregado, definidos como atributos do nó correspondente. A cada nó que contém filhos, fora ainda definido um dos campos da projeção da consulta *SQL* para que seja repassado como parâmetro para seus nós-filhos. A Figura 30 ilustra o esquema LICITACAO_FASE1, dispostos na interface implementada.

Figura 30 - Interface da aplicação



Fonte: Autor

Após o pré-processamento dos esquemas, definição das consultas e associação entre campos e atributos, o algoritmo recursivo *JSONDocGenerate*, apresentado na Seção 4.3, foi executado. Este algoritmo processa em cadeia a execução de todas as consultas, na sequência exata em que os nós do esquema se encontram na árvore hierárquica. Os resultados do processamento foram documentos na notação JSON, instanciados exatamente como previstos nos esquemas carregados. A Figura 31 apresenta um trecho de um documento *JSON* gerado a partir do esquema LICITACAOFASE1.

Figura 31 - Trecho de um documento JSON gerado

Fonte: Autor.

Os *Esquemas* JSON utilizados, bem como um documento JSON gerado a partir de cada esquema, encontram-se ao término desta dissertação.

Os documentos JSON resultantes do processamento totalizaram 2120, sendo estes: 814 instâncias do esquema LICITACAOFASE1, e 701 referentes a o esquema LICITACAOFASE2.

A diferença entre o número de documentos gerados, e o total de registros processados, descritos na Seção 5.2, deve-se à existência de processos cadastrados de forma incompleta no banco relacional, estes foram desconsiderados no estudo realizado.

6 CONCLUSÃO

A necessidade crescente de informações dispostas em documentos semiestruturados, capazes de serem interpretados por sistemas externos, vem impulsionando a utilização de notações abertas e independentes de linguagem como o JSON. Seguindo está tendência, recentemente alguns órgãos fiscalizadores, como o Tribunal de Contas dos Municípios do Estado de Goiás, adotaram JSON como o formato exigido para o recebimento das informações necessárias à prestação de contas. A adoção de notações abertas como formato preferencial para o intercâmbio de informações contribui sobremaneira para a disseminação de informações legível e transparentes, porém obtê-las através de dados armazenados em bancos de dados relacionais, principalmente originadas de sistemas legados representa um desafio. O processo proposto neste trabalho demonstrou eficiência na geração de tais documentos. A versatilidade do processo permite que os documentos sejam gerados para múltiplos propósitos, com a estrutura e escopo fieis aos previstos em esquemas JSON válidos. A geração dos documentos *JSON* é realizada seguindo estritamente a estrutura determinada pelo esquema previamente carregado e mapeamento de forma automática. As consultas *SQL* responsáveis pela obtenção dos dados utilizam instruções padrão, garantindo compatibilidade com a maioria dos bancos relacionais atualmente utilizados. Desta forma, a obtenção dos resultados esperados independe da existência de suporte nativo à notação JSON, tanto por parte da linguagem utilizada, como também do banco de dado relacional de origem dos dados.

Como a proposta desenvolvida neste trabalho tratou do mapeamento dos esquemas JSON e geração de documentos JSON para múltiplos propósitos, foram trabalhadas as propriedades necessárias a estruturação dos dados. No entanto, poderão existir nos esquemas outras propriedades destinadas à validação dos valores representados. Tais propriedades poderão ser consideradas em trabalho futuro. Neste trabalho de dissertação, para que fosse possível o mapeamento automático dos esquemas JSON, foi necessário resolver as referências entre as propriedades e suas definições. Porém limitou-se às definições anotadas no próprio esquema analisado. Um trabalho futuro poderá explorar a resolução dos referenciamentos que poderão ocorrer, embora menos frequentes, entre esquemas JSON distintos. Em um trabalho futuro, o processo proposto poderá ser utilizado para a migração de bancos de dados relacionais para bancos NoSQL orientados a documentos. A vantagem de permitir a definição de escopo, por meio dos esquemas mapeados, possibilita extrair das tabelas relacionais apenas as informações que sirvam aos interesses do banco NoSQL resultante.

REFERÊNCIAS BIBLIOGRÁFICAS

ASAI, T.; ABE, K.; KAWASOE, S. A. H.; SAKAMOTO, H.; ARIKAWA, S. **Efficient substructure discovery from large semi-structured data**. IEICE TRANSACTIONS on Information and Systems. 2004.

AZEVEDO, M. I. M.; AMORIM, L. P.; ZIVIANI, N. **A universal model for XML information retrieval**. LNCS. Springer, Heidelberg. 2005.

BEN-KIKI, O.; EVANS, C.; INDY, D. N. **YAML Ain't Markup Language**. Version 1.2. 3rd Edition. 2009.

BROCKE, J. V.; ROSERMAN M. **Manual de BPM: gestão de processos de negócio**. Bookman. 2010.

CAFEZEIRO, I; VITERBO, J; RADEMAKER, A; ENDLER, M; HAEUSLER, E. H. **"A formal Framework for modeling context-aware behavior in ubiquitous computing" Communications in Computer and Information Science**, V.17, 2008.

CODD, E. F. **A Relational Model of Data for Large Shared Data Banks**. Revista CACM. 1970.

COHEN, W. **Integration of heterogeneous databases without common domains using queries based on textual similarity**. In Proc. ACM SIGMOD, Seattle, WA. 1998.

CROCKFORD, D. JSON: **The Fat-Free Alternative to XML**. Presented at XML 2006 in Boston, December 6, http://www.JSON.org/fatfree.html. Acesso em: 27/08/2017

DE PAIVA, E.; REVOREDO, K. **Big Data e Transparência: Utilizando Funções de Mapreduce para incrementar a transparência dos Gastos Públicos**. XII Simpósio Brasileiro de Sistemas de Informação (SBSI), 2016.

ELMASRI, R.; NAVATHE. **Sistemas de Banco de Dados: Fundamentos e Aplicações**. Pearson Education, 2006.

Guia de Transparência Ativa Para Órgãos e entidades do Poder Executivo Federal. 5a versão. Controladoria Geral da União (CGU). Disponível em: <http://www.acessoainformacao.gov.br/lai-para-sic/sic-apoio-orientacoes/guias-e-orientacoes/gta-5a-versao.pdf>. Acesso em: 08 out. 2018.

GOLDBERG, K. H. XML: **Visual QuickStart Guide, Second Edition**. Peachpit Press, 2009

IBM DB2 DATABASE DOCUMENTATION.
https://www.ibm.com/analytics/br/pt/technology/db2 Acessado em: Agosto de 2018

IMHOF, R.; FROZZA, A. A.; MELLO, R. S. **Um Survey sobre Extração de Esquemas de Documentos JSON**. In: ESCOLA REGIONAL DE BANCO DE DADOS (ERBD), 1/2017.

XIII Escola Regional de Banco de Dados 2017 (ERBD 2017). Porto Alegre: Sociedade Brasileira de Computação, apr. 2017 . ISSN 2595-413X.

ISSARNY, V.; GEORGANTAS, N.; HACHEM, S.; ZARRAS, A.; VASSILIADIS, P.; AUTILI, M.; et al. **Service-Oriented Middleware for the Future Internet: State of the Art and Research Directions**. Journal of Internet Services and Applications, Springer, 2011. Disponível em: https://hal.inria.fr/inria-00588753v1/document Acesso em: Janeiro de 2019

JATANA, N.; PURI, S.; AHUJA, M.; KATHURIA, I.; GOSAIN, D. **A survey and comparison of relational and non-relational database. International Journal of Engineering Research and Technology 2012**; 1(6):1–5. ISSN 2278-0181 Disponível em: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.678.9352&rep=rep1&type=pdf Acessado em: Janeiro de 2019

JELLEMA, L. **Creating JSON document straight from SQL query - using LISTAGG and With Clause**. https://technology.amis.nl/2011/06/14/creating-json-document-straight-from-sql-query-using-listagg-and-with-clause/ Acesso em: Maio de 2018

JÚNIOR, J. B. da S.; SILVA, P. C. **Análise da representação semântica de modelos de dados do formato JSON**. Programa de Pós-Graduação em Sistemas e Computação – Universidade Salvador, BA Revista de Sistemas e Computação, v. 8, n. 1, p. 196-209, jan./jun. 2018

KARNITIS, G.; ARNICANS, G. **Migration of relational database to document-oriented database: structure denormalization and data transformation**. In Proceedings of the International Conference on Computational Intelligence, Communication Systems and Network, 2015.

KONININ, L. **LKJSON - JSON delphi library**. https://sourceforge.net/projects/lkJSON Acessado em: Junho de 2018

LEI Nº 12.527, de 18 de novembro de 2011. Diário Oficial da República Federativa do Brasil, Brasília, DF, 18 nov. 2011. Disponível em: <http://www.planalto.gov.br/ccivil_03/_Ato2011-2014/2011/Lei/L12527.htm>. Acesso em: 08 out. 2018.

Manual de Dados Abertos: Governo. W3C (World Wide Web Consortium). Disponível em: <http://www.w3c.br/pub/Materiais/PublicacoesW3C/Manual_Dados_Abertos_WEB.pdf>. Acesso em: 08 jun. 2018.

MCGHAN, D. **Relational to JSON with SQL**. https://jsao.io/2018/10/relational-to-json-with-sql Acesso em: Nov. 2018

ORACLE DATABASE DOCUMENTATION. JSON in Oracle Database. Oracle Database Online Documentation Library, 12c Release 1 (12.1.0.2). https://docs.oracle.com/database/121/ADXDB/json.htm#ADXDB6246 Acesso em: Ago. 2018

PETKOVIĆ, D. **SQL/JSON Standard: Properties and Deficiencie**. University of Applied Sciences Hochschulstr, Rosenheim, Germany, 2017.

PETKOVIĆ, D. **JSON Integration in Relational Database Systems**. IJCA, Int. Journal of Computer Applications, Vol. 166, No.13, 2017.

Portal do Tribunal de Contas dos Municípios do Estado de Goiás. Disponível em: <https://www.tcm.go.gov.br/site/o-tcm/>. Acesso em: 28 jul. 2018.

Prestação Eletrônica de Contas do Tribunal de Contas dos Municípios do Estado de Goiás. Disponível em: <https://testes.tcm.go.gov.br/colare-doc/faces/public/pagInicio.xhtml?dswid=-7403>. Acesso em: 05 ago. 2018.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. 2nd ed. [S.l.]: Prentice Hall, 2002.

SANTOS T. F. P. **SecJSON - Secure Javascript Object Notation**. Dissertação de Mestrado. ISCTE-IUL, Lisboa/Portugal. 2016.

SANTOS, K. C. P. **Utilização de ontologias de referência como abordagem para interoperabilidade entre sistemas de informação utilizados ao longo do ciclo de vida dos produtos**. 2011. 101 f. Dissertação (Mestrado em Engenharia Mecânica e de Materiais) – Universidade Tecnológica Federal do Paraná, Curitiba, 2011.

SETZE, V.; SILVA, F. S. **Banco de Dados: Aprenda o que são, melhore seu conhecimento, construa os seus**. São Paulo: Edgard Blücher, 2005.

SILVERIA, V. N. K. **X-Spread : um mecanismo automático para propagação da evolução de esquemas para documentos XML**. Dissertação de Mestrado. UFRGS. 2007.

STRAUCH, C. **NoSQL Databases**. Fevereiro 2011. Disponível em : http://www.christofstrauch.de/nosqldbs.pdf Acessado em : janeiro de 2019

TERLSON, B. **ECMAScript® 2018 Language Specification**. 9º edição (ECMA 262). 2018. http://www.ecma-international.org/ecma-262/9.0 Acessado em 01/07/2018

The home of Esquema JSON. Disponível em: <https://json-schema.org/>. Acessado em: janeiro de 2018.

WRIGHT, A.; ANDREWS, H. **Esquema JSON**. Cloudflare, Inc. São Francisco, CA, EUA. 2018. http://JSON-schema.org/. Acessado em 20/06/2018.

ZHAO, G.; LIN, Q.; LI, L.; LI, Z. **Schema conversion model of SQL databasto nosql**. In P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), Ninth International Conference. IEEE. 2014.

ANEXO

(Protótipo Implementado)

De modo a validar o processo proposto no capítulo 3, fora implementado um protótipo funcional que executa os procedimentos sugeridos e retorna os resultados esperados.

Em conformidade com o principal objetivo deste trabalho, propor um meio eficiente e ágil para a geração automatizada de documentos JSON, uma interface intuitiva que proporcione boa usabilidade é de fundamental importância. A ferramenta implementada, denominada *JSONUncGen*, responsável pela execução dos procedimentos descritos no capítulo anterior, fora desenvolvida utilizando a *IDE Embarcadero Delphi*, para ambiente Windows. O Sistema Gerenciador de Banco de Dados (SGBD) utilizado fora o IBM DB2 (IBM DB2 *online Documentation*), porém os processos propostos utilizam *SQL Scripts* compatíveis com a maioria dos bancos de dados relacionais mais conhecidos.

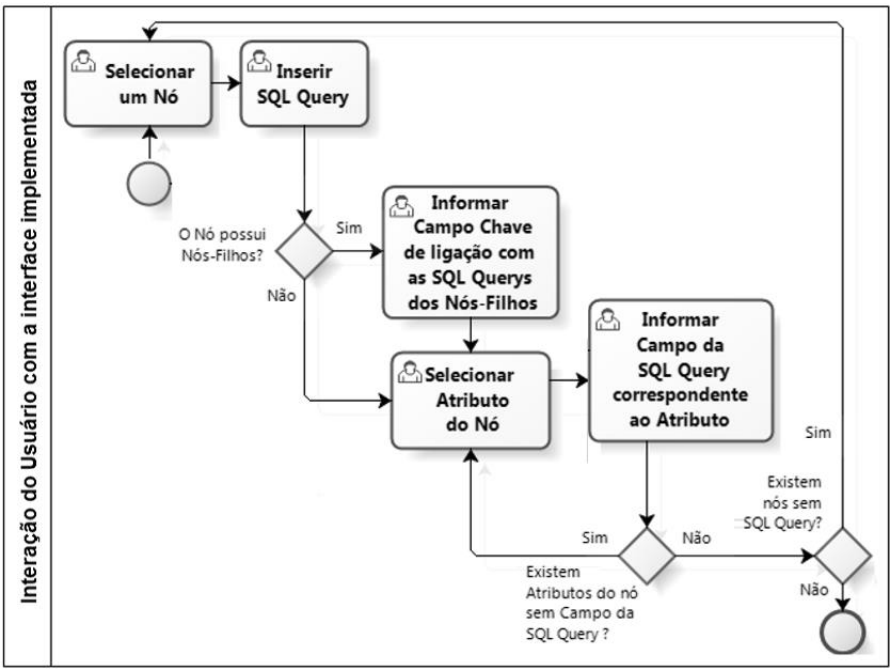
Para a carga inicial do esquema, utilizou-se a biblioteca *LkJSON* (*JSON delphi library* Versão 1.07), está biblioteca permite a manipulação de documentos estruturados na notação JSON. Tendo em vista que esquemas JSON são também documentos JSON válidos, a classe "*TlkJSONstreamed*", mais especificamente os métodos "*loadfromfile*" e "*GenerateReadableText*" foram utilizados, possibilitando respectivamente a carga inicial do esquema e a endentação dos documentos JSON obtidos por meio deste trabalho.

Embora a *LkJSON* tenha sido utilizada para a carga do esquema, a mesma não provê recursos suficientes para a resolução de referências entre as propriedades do esquema e suas respectivas definições. Para a solução deste problema foram implementados neste trabalho métodos que permitem o tratamento destas referências. Todos os algoritmos, descritos em detalhes nas seções seguintes, foram desenvolvidos para este trabalho de dissertação.

A análise dos esquemas JSON, quando realizada de forma manual, exige tempo e dedicação por parte dos profissionais envolvidos. Esquemas contendo um grande número de vetores e objetos complexos, muitas vezes poderá tornar difíceis os processos de geração dos documentos instancias destes esquemas. Em cenários como este, a ferramenta proposta neste trabalho poderá auxiliar a reduzir os custos destas tarefas. Por possibilitar a carga dos esquemas, e o mapeamento destes de forma automatizada, possibilita uma redução considerável do tempo necessário à obtenção dos resultados esperados.

O diagrama BPMN a seguir (Figura 32) representa a interação do usuário com a interface do protótipo implementado.

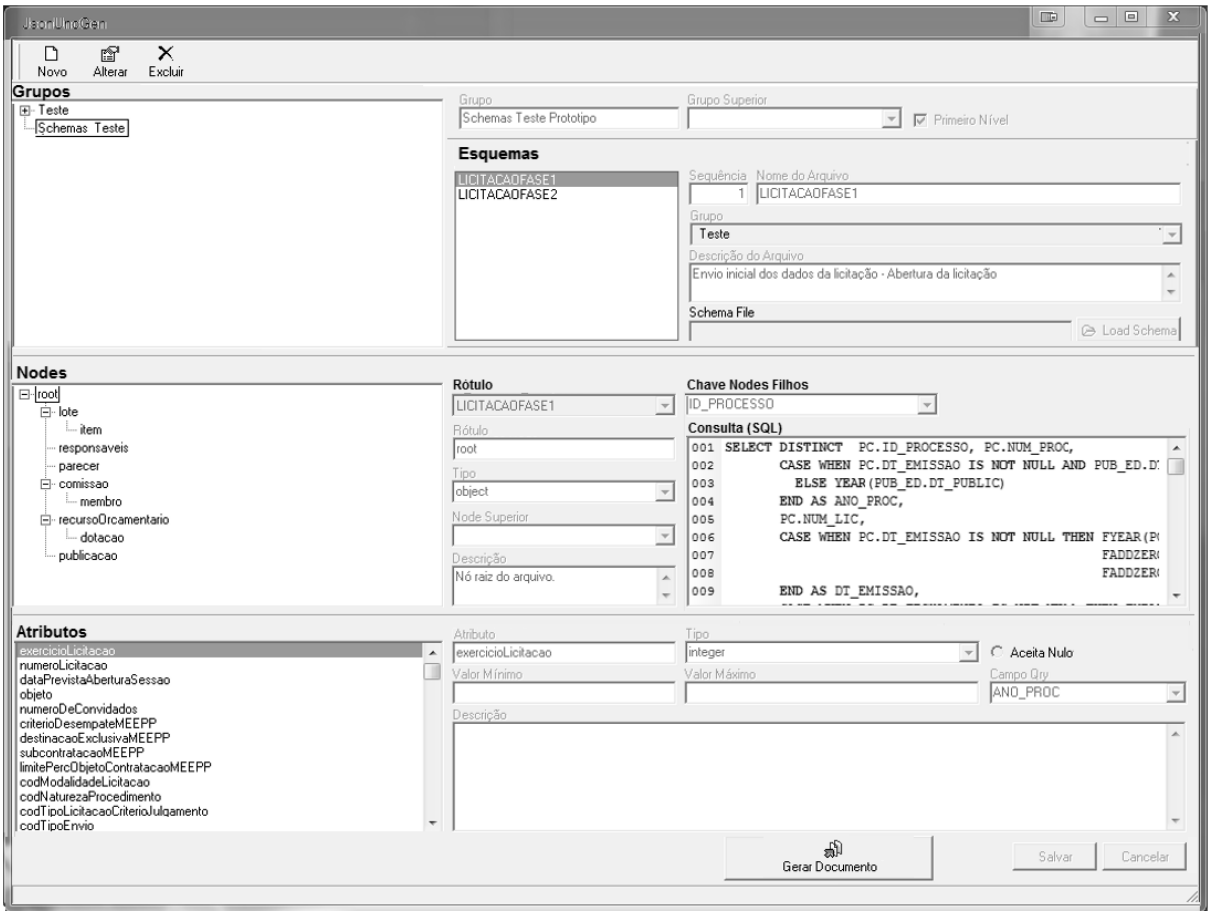
Figura 32 - Interação com a interface implementada



Fonte: Autor.

A Figura 33 ilustram a tela principal do protótipo JSONUncGen.

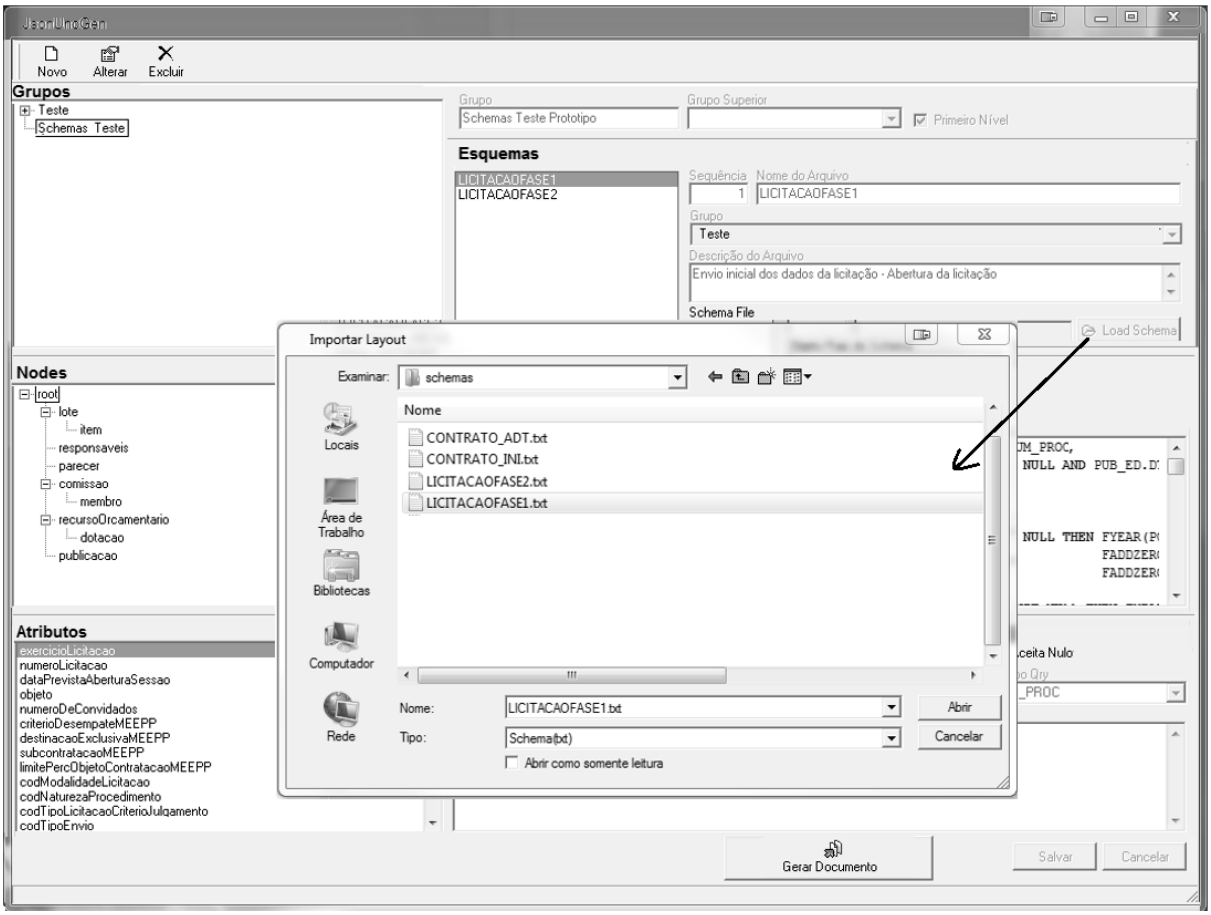
Figura 33 – Tela principal de JSONUncGen



Fonte: Autor.

A seleção do esquema *JSON* a ser carregado é realizada pelo usuário ao acionar o botão *Load Schema* (Figura 34).

Figura 34 – Seleção do esquema JSON a ser carregado



Fonte: Autor.

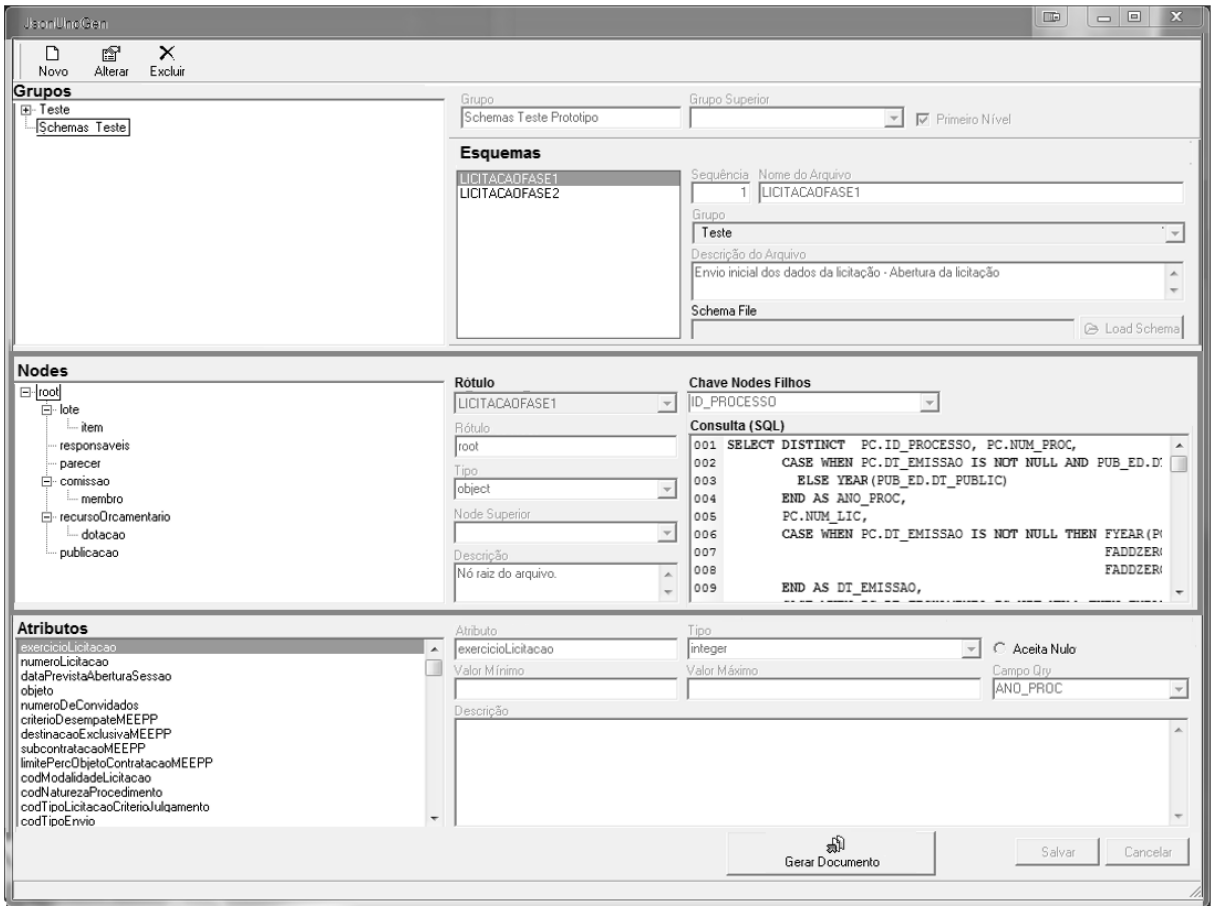
Ao salvar o registro, a aplicação solicitará a confirmação do usuário quanto a importação do esquema selecionado. Após a confirmação, todos os campos do esquema são mapeados e inseridos no banco relacional de forma automática.

A tela principal da aplicação apresenta no lado esquerdo uma lista hierárquica de grupos. A possibilidade de cadastrar novos grupos, através do preenchimento do rótulo do Grupo e da seleção do Grupo Superior (opcional), auxilia na organização dos esquemas *JSON*, que poderão ser agrupados por domínio ou outro critério definido pelo usuário.

Ainda nesta tela foram adicionadas três regiões: Esquemas, Nodes e Atributos. Na região Esquemas estão concentradas todas as informações destes. Ao cadastrar um novo esquema, um documento esquema JSON poderá ser informado, neste caso, ao salvar o esquema, a aplicação apresentará uma caixa de diálogo solicitando confirmação para que seja realizada a carga e mapeamentos do esquema informado. Caso haja a confirmação para a execução da carga

do esquema, todas as informações dos Nós e Atributos originadas do esquema serão "alimentadas" automaticamente. A aplicação também permite o cadastramento manual de todas as informações, este recurso é útil quando não há um esquema JSON para carregar. A Figura 35 destaca os campos dos nós do esquema.

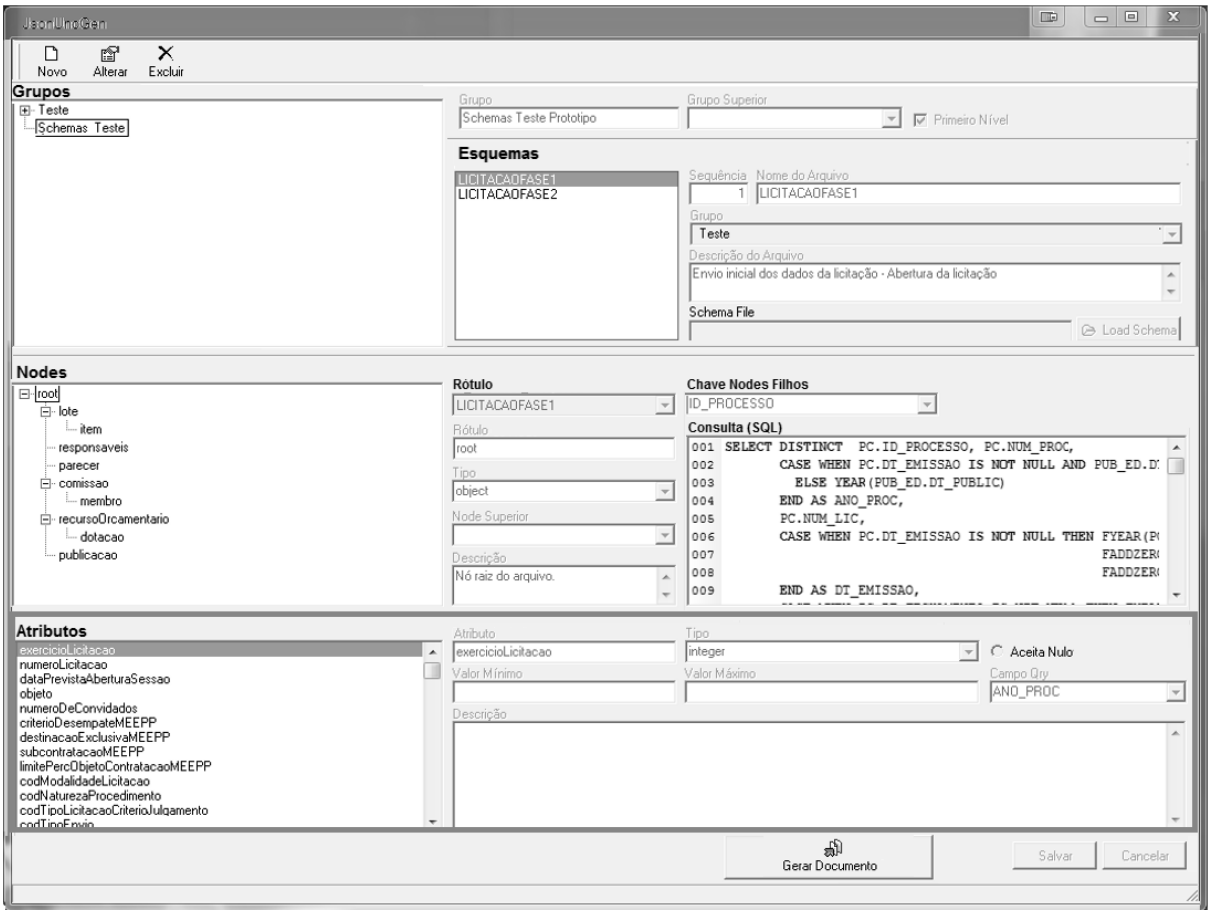
Figura 35 – Interface de JSONUncGen - Nodes



Fonte: Autor.

A região *Nodes* apresenta as informações dos nós mapeados por meio do processo de carga do esquema JSON, ou cadastrados manualmente. Observa-se que os nós se encontram dispostos em árvore hierárquica, para cada um dos nós deve ser informada uma consulta *SQL* responsável pela recuperação dos dados que serão associados aos atributos dos nós. Para os nós que tiverem filhos, o preenchimento do campo "Chave *Nodes* Filhos" é obrigatório, este será selecionado dentre os campos projetados na consulta, o valor deste campo será repassado como parâmetro para as consultas dos nós-filhos, permitindo que durante a geração dos documentos JSON, a execução de tais consultas ocorra de forma automática e sequencial.

Figura 36 – Interface de JSONUncGen - Atributos

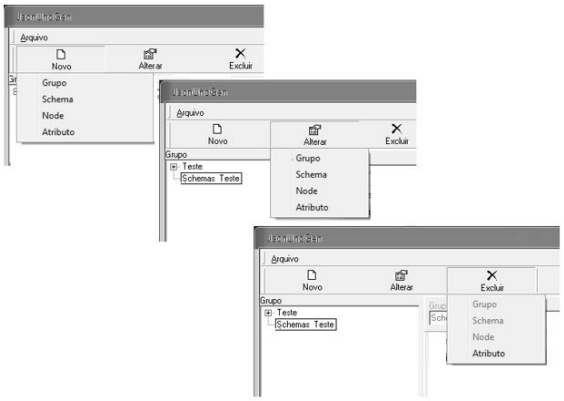


Fonte: Autor.

A região Atributos (Figura 36) concentra as informações de todas as propriedades simples que compõem o esquema correspondente ao documento *JSON* que será gerado. A lista de atributos mantém a sequência exata com que os mesmos se encontram anotados no documento de esquema carregado. O campo “Campo Qry” deverá ser selecionado dentre os campos projetados na consulta *SQL* associada ao nó ao qual o atributo pertence. Este campo é de fundamental importância pois este originará o valor do atributo a ser anotado no documento *JSON* que será gerado.

A interface de JSONUncGen permite o cadastramento manual de todos os elementos do esquema. Este recurso torna possível a geração de documentos mesmo nos casos em que não há esquema para ser carregado.

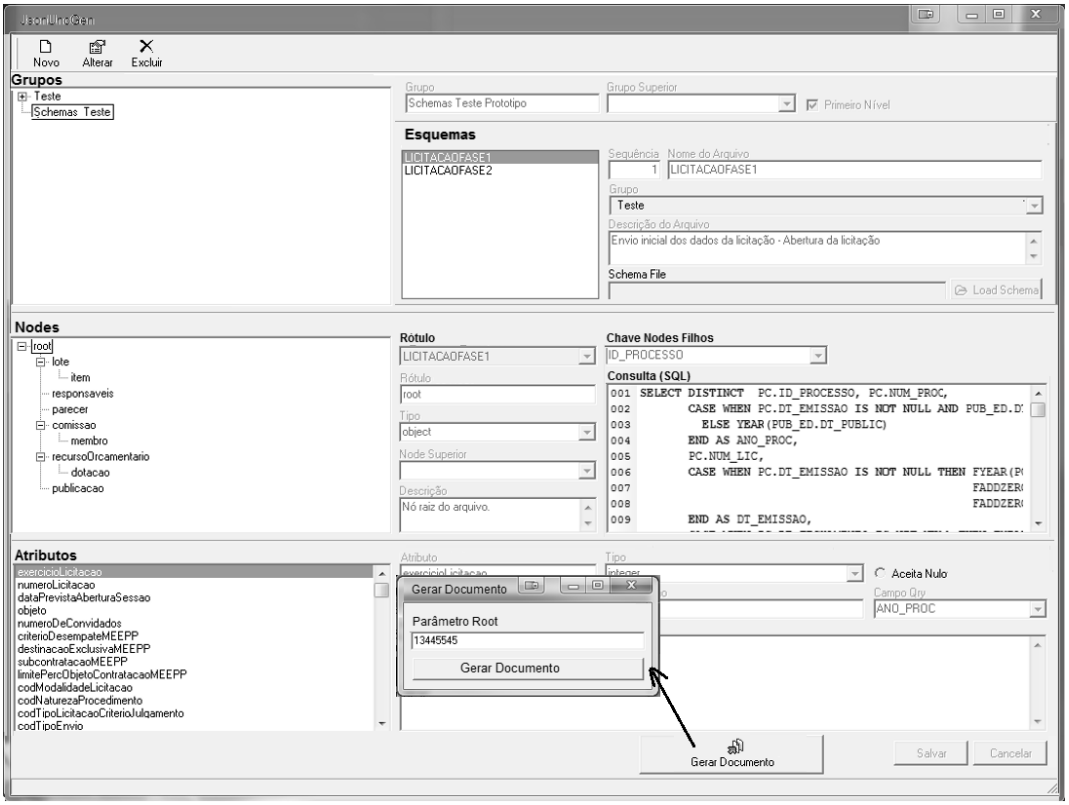
Figura 37 – Barra de ferramentas - opções para cadastro manual dos esquemas



Fonte: Autor.

Através dos botões “Novo”, “Alterar” e “Excluir” dispostos na barra de ferramentas superior (Figura 37), é possível manipular o cadastro de todos os elementos do esquema.

Figura 38 – Tela de passagem de parâmetro para o nó raiz



Fonte: Autor.

Para que a execução sequencial do processo de geração do documento *JSON* inicie, é necessário informar o parâmetro do nó raiz do esquema (Figura 38).